UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

Département d'informatique et d'ingénierie

Ce mémoire intitulé :

RECHERCHE DE TROUS NOIRS DANS LES RÉSEAUX INFORMATIQUES À L'AIDE D'AGENTS MOBILES

présenté par
Eric Vachon

pour l'obtention du grade de maître ès science (M.Sc.)

a été évalué par un jury composé des personnes suivantes :

Dr. Andrzej Pelc .... Directeur de recherche
Dr. Rokia Missaoui .... Presidente du jury
Dr. Jurek Czyzowicz .... Membre du jury

Mémoire accepté le : 28 juillet 2006

À ma famille et mes amis.

Remerciements

À Andrzej Pelc pour son soutien et ses conseils tout au long de ce mémoire.
Au CRSNG et à la chaire de recherche CALDI pour le soutien financier.
À Sophie et à Marie pour leur aide à la correction du français de ce mémoire.
À Moe et à Samuel pour leurs conseils.
À ma famille et mes amis pour leurs encouragements.

Merci !

2.2.1 Graphes non dirigés avec étiquettes 13
2.2.2 Graphes dirigés avec étiquettes 14
2.2.3 Graphes sans étiquettes 15
2.2.4 Restrictions sur l'agent 16
2.2.5 Graphes avec panne sur les liens 17
2.3 Problème des trous noirs 18
2.3.1 Modèle asynchrone 20
2.3.2 Modèle synchrone 22

3 Principes méthodologiques et propriétés de base 25
3.1 Méthodologie 25
3.2 Modèle et terminologie 26
3.3 Résultats généraux pour les graphes 27

4 Résultats pour l'anneau 31

5 Résultats pour le tore de grandeur 2 par k 44

6 Résultats pour le tore de grandeur m par k 58

7 Conclusion 69

A Code de anneau.java 70

Bibliographie 73

Liste des figures

2.1 La diffusion générale et l'échange total 10

3.1 La procédure *sonder* 29

3.2 La procédure *fourche* 30

5.1 Tore de grandeur 2 par 6 44

5.2 Tore 2 par k : Exemples d'explorations 47

5.3 Tore 2 par k : Deux premières phases 49

5.4 Tore 2 par k : Configuration d'exploration 50

5.5 Tore 2 par k : Dernière phase de l'exploration 50

5.6 Tore 2 par k : Configurations possibles 51

5.7 Tore 2 par k : Deuxième configuration, dernière phase 53

5.8 Tore 2 par k : Induction sur la troisième configuration 54

5.9 Tore 2 par k : Troisième configuration, avant dernière phase 55

5.10 Tore 2 par k : Dernière phase pour les configurations 4,5 et 6 56

6.1 Tore m par k : Séquence des phases : légère, normale, légère 60

Liste des tableaux

4.1 Anneau avec territoire connu de 7 ou 8 noeuds 41

4.2 Temps pour les anneaux $n \leq 7$ 42

6.1 Borne inférieure sur tore de grandeur m par k 63

6.2 Temps de l'algorithme Tore 67

6.3 Algorithme Tore : Différence avec la borne inférieur en pire cas 68

Résumé

La technologie des agents mobiles est un nouveau paradigme qui permet à des logiciels de suspendre leur exécution sur un hôte, de se transporter sur un autre hôte et de reprendre l'exécution au nouvel endroit. Cette technologie peut être utilisée, par exemple, dans une application commerciale ou pour effectuer un calcul distribué. La sécurité de ces agents devient une préoccupation importante dans les réseaux. Une menace pesant sur les agents est la présence de *trous noirs*, qui sont des hôtes qui détruisent complètement les agents qui les visitent. Ce mémoire présente des algorithmes de recherche de trous noirs dans les réseaux informatiques, ayant une topologie de tore, à l'aide d'agents mobiles partiellement synchrones. Des algorithmes optimaux ont été trouvés pour les anneaux, les tores de grandeur 2 par k, ainsi que les tores de grandeur 3 par k. Dans le cas général nous présentons un algorithme d'approximation avec taux 1.3 qui est asymptotiquement optimal lorsque la plus petite dimension du tore n'est pas bornée.

Abstract

The technology of mobile agents is a new paradigm that makes it possible for software to suspend its execution on a host, to transfer itself onto another host and to continue its execution at the new location. This technology can be used, for example, in a commercial application or to carry out a distributed computation. The safety of these agents becomes an important concern in networks. One of the threats that the agents face is the presence of *black holes*, which are hosts that completely destroy the agents visiting them. This thesis presents algorithms in the data-processing networks, having a torus topology by using partially synchronous mobile agents. Optimal algorithms were found for the rings, the tori of sizes 2 by k, as well as the tori of sizes 3 by k. In the general case we present an approximation algorithm with a 1.3 rate which is asymptotically optimal when the smallest dimension of the torus is not limited.

Chapitre 1

Introduction

La technologie des agents mobiles est un nouveau paradigme qui permet à des logiciels de suspendre leur exécution sur un hôte, de se transporter sur un autre hôte et de reprendre l'exécution au nouvel endroit. Les agents aujourd'hui peuvent être de divers types, allant de simples objets distribués à des logiciels organisés combinant l'intelligence artificielle.

Une des préoccupations principales des systèmes utilisant les agents mobiles doit être la sécurité. Il est important de protéger à la fois les hôtes et les agents des menaces pouvant provenir de leur environnement [20]. Ce problème est particulièrement pressant dans un environnement peu coopératif où il existe peu de normes, tel Internet. En fait, même si les agents sont dans un environnement contrôlé où ils coopèrent à un but commun, comme par exemple dans le cas d'un calcul distribué, des menaces au niveau de la sécurité peuvent survenir. Un bris matériel ou une défaillance de logiciel peuvent, par exemple, rendre un hôte dangereux pour les agents.

Notre problème concerne la sécurité des agents dans les réseaux informatiques. Nous allons premièrement décrire ce que sont les agents et quels sont leurs avantages. Ensuite, nous allons expliciter le problème à la solution duquel nous voulons contribuer avec cette recherche.

1.1 Les agents mobiles

Le concept d'agents mobiles provient d'une analyse critique des méthodes de communication réalisée depuis la fin des années 1970. Cette section fait l'état des résultats de cette analyse et présente la notion des agents mobiles.

1.1.1 Approche classique

La communication dans les réseaux informatiques utilise l'*appel de procédure à distance* (RPC). Conçu vers la fin des années 1970, ce paradigme voit la communication dans ces réseaux comme étant la possibilité qu'un des systèmes puisse invoquer des procédures sur l'autre système. Chaque message sur le réseau transporte soit une requête soit le résultat d'un de ces appels. La requête contient les données qui sont les paramètres d'entrée de la procédure appelée et la réponse contient les données produites par la procédure appelée. Les systèmes qui établissent une telle communication s'entendent à l'avance sur les différentes procédures accessibles, le type d'arguments utilisé et les résultats obtenus par ces procédures. Cette entente s'appelle le *protocole* [23].

Pour effectuer des tâches distantes, un utilisateur doit envoyer à un système distant une série d'appels de procédure à distance. Chaque appel représente un message qui circule sur le réseau. Par exemple, un utilisateur voulant supprimer tous les fichers de plus de deux mois sur un serveur distant doit envoyer un message pour avoir la liste des fichiers, attendre la réponse, sélectionner les n fichiers datés de plus de deux mois et envoyer un message par fichier à supprimer. De plus, pour des traitements plus complexes et plus longs, pour avoir une interaction continue, il est nécessaire de maintenir une communication constante. Ceci est peu pratique lorsque le coût de la communication est élevé.

1.1.2 Approche par agents mobiles

Une procédure alternative est d'utiliser la *programmation distante.* Cette approche permet, non seulement, d'appeler des procédures sur un système distant, mais de lui fournir les procédures à exécuter. Chaque message sur le réseau transporte une procédure que le système hôte doit exécuter ainsi que les données associées à cette procédure. Celle-ci est mise en veille sur le système appelant et est envoyée sur un système hôte afin qu'elle

poursuive son exécution. Les données représentent l'état de la procédure, c'est-à-dire à quel endroit de son exécution elle doit reprendre sur le système hôte.

Les deux systèmes s'entendent sur les instructions qui sont autorisées dans les procédures et sur les types de données qui sont alloués dans les états. Cette entente s'appelle le *langage*. Le langage inclut les instructions qui permettent aux procédures de prendre des décisions, d'examiner et de modifier leurs états et d'appeler des procédures disponibles sur le système hôte. Ces procédures et leurs états sont appelés *agents mobiles* pour mettre l'emphase sur le fait qu'ils représentent le système qui les a créés, même lorsqu'ils sont dans le système hôte [23].

Un utilisateur voulant supprimer tous les fichiers de plus de deux mois sur un serveur distant n'a qu'un message à envoyer sur le réseau, l'agent pourra décider sur place quels fichiers sont à supprimer. Les agents peuvent aussi être utilisés pour un calcul distribué complexe. Par exemple, lorsqu'un processus se rend à un point où il peut effectuer un calcul en parallèle, il crée des agents qui vont effectuer leur partie de calcul sur des systèmes hôtes pour ensuite revenir donner les résultat au processus parent. On peut même penser à une utilisation d'applications sur demande, les agents allant effectuer un travail sur les systèmes hôtes pour un certain temps et étant supprimés par la suite.

Un avantage de l'approche par agents mobiles est la performance au niveau de la communication sur le réseau : plus la tâche est longue et complexe, plus les agents offrent un avantage sur le nombre de messages envoyés sur le réseau. Non seulement on fait une économie sur le nombre de messages, mais la communication n'a pas besoin d'être établie en permanence lors du traitement. Cet avantage peut être très important lorsqu'on ne peut pas se permettre une communication constante entre les systèmes. Un autre bénéfice est qu'on peut personnaliser le traitement à effectuer sur le système hôte, en définissant des procédures spécifiques pour des traitements spécifiques. Ainsi, l'utilisateur final n'a plus besoin d'installer plusieurs applications localement pour bénéficier des services d'une entreprise, la tâche étant accomplie par les agents dans un environnement contrôlé.

1.2 Définition du problème

La nature des agents mobiles les rend vulnérables aux attaques des hôtes sur le réseau. Un agent qui est compromis ou même pire, détruit, peut interrompre un traitement qui

a été amorcé précédemment et l'empêche de se terminer correctement. Il devient donc important de développer des stratégies pour protéger les agents.

Le plus grand danger que court un agent est de rencontrer un *trou noir*, c'est-à-dire un hôte qui détruit complètement un agent qui le visite sans laisser de traces. Un tel danger n'est pas improbable et peut être causé par les défaillances matérielles ou par les virus infectant des hôtes du réseau. Pour protéger les agents d'une telle menace, des algorithmes de recherche de trou noir ont été proposés dans la littérature (section 2.3). Une fois la localisation des trous noirs identifiée, les agents peuvent éviter ces hôtes dangereux ou bien des actions de récupérations peuvent être entreprises.

Certains chercheurs ont proposé un modèle où les agents sont de nature asynchrone (section 2.3.1), c'est-à-dire que le déplacement des agents se fait en un temps fini, mais dont la durée n'est pas bornée. Une telle supposition implique qu'il nous est impossible de déterminer avec certitude si un agent a rencontré un trou noir ou bien s'il traverse simplement un lien qui est lent.

Le cas des agents totalements asynchrones survient rarement en pratique. On peut souvent établir une borne supérieure sur le temps de déplacement des agents. Il devient donc intéressant d'étudier la recherche de trous noirs dans cet environnement qui est dit partiellement synchrone. La recherche (section 2.3.2) dans ce domaine ne fait que débuter, certains résultats pour les réseaux à topologie de ligne et de l'arbre ont été trouvés [7] et des études sur la complexité [8, 21] ont été faites.

Nous proposons de continuer les travaux sur la recherche de trou noir dans les réseaux informatiques à l'aide d'agents mobiles dans le modèle partiellement synchrone. Notre travail porte sur les réseaux à topologie de tore comportant un trou noir et utilisant deux agents mobiles. Deux est le nombre minimal d'agents pour localiser un trou noir : un agent a été prouvé insuffisant. La topologie du tore a été choisie à cause de sa simplicité et sa popularité dans les applications. Un cas particulier du tore est l'anneau : une des architectures fréquemment rencontrées en pratique. À la lumière de l'utilité des agents mobiles et du besoin actuel au niveau de la sécurité, nous pensons que notre travail sera utile pour la communauté informatique.

1.3 Présentation des résultats

Nous avons réussi à obtenir des algorithmes optimaux du point de vue du temps de fonctionnement pour les anneaux, ainsi que les tores de grandeur 2 par k et les tores de grandeur 3 par k. Dans le cas général nous présentons un algorithme d'approximation avec taux 1.3 qui est asymptotiquement optimal lorsque la plus petite dimension du tore n'est pas bornée.

La plus grande partie du travail de ce mémoire est de trouver les bornes inférieures pour le temps des algorithmes travaillant dans chacune des topologies étudiées. Les algorithmes d'exploration efficace viennent de manière presque naturelle lorsque l'on connaît bien le sujet. C'est de prouver qu'ils sont optimaux ou asymptotiquement optimaux qui est la difficulté principale de ce mémoire.

Chapitre 2

Revue des écrits

Cette revue de la littérature comprend trois sections. Puisque le sujet de la recherche d'un trou noir dans les réseaux informatiques à l'aide d'agents mobiles est récent, il est intéressant d'en voir les sources et les domaines connexes. Nous allons faire un bref survol des différents problèmes et modèles que l'on peut rencontrer dans les deux branches qui forment notre sujet. La première de ces branches est la tolérance aux pannes dans les réseaux. En effet, les trous noirs sont un type de panne qui agit sur les agents circulant dans les réseaux. La deuxième branche est celle de l'exploration des graphes à l'aide d'agents mobiles. La recherche des trous noirs dans les réseaux se rapproche aussi de l'exploration des graphes puisque des agents doivent parcourir un réseau qui est représenté sous la forme d'un graphe. Finalement nous allons entrer dans le vif du sujet en présentant ce qui a été fait jusqu'à présent dans le domaine de la recherche d'un trou noir dans les réseaux à l'aide d'agents mobiles.

2.1 Tolérance aux pannes dans les réseaux informatiques

Plus les réseaux grandissent, plus ils deviennent vulnérables aux pannes matérielles. Certains liens et/ou noeuds du réseaux peuvent tomber en panne. Il devient alors essentiel de développer des algorithmes pouvant assurer une communication efficace malgré ces pannes, souvent sans même connaître leur localisation. Il existe deux stratégies pour assurer une communication efficace en présence de pannes : utiliser des algorithmes de

communication tolérants aux pannes ou faire le diagnostic des pannes pour ensuite les éviter. La première stratégie propose de faire passer les messages dans le réseau sans savoir à l'avance où sont les pannes. Elle utilise plutôt la redondance pour obtenir des résultats. La deuxième, le diagnostic des pannes, utilise des tests pour déterminer la localisation des pannes dans le réseau (ou dans un système multiprocesseur). La communication peut ensuite être faite en évitant les processeurs en panne ou en les remplaçant par des processeurs fonctionnels. Certains paramètres du modèle doivent être déterminés : le mode de communication, le type de pannes, la durée des pannes, la distribution des pannes et la flexibilité de l'algorithme. La définition de ces paramètres présentés ici a été tirée de [28] et [19]. Nous présenterons ensuite la communication tolérante aux pannes ainsi que le diagnostic des pannes.

Les réseaux informatiques analysés sont représentés par des graphes, dont les noeuds représentent les hôtes et les arêtes représentent des liens de communication. Si l'arête est dirigée, la communication ne peut se faire que du noeud situé à la queue de l'arc au noeud situé à la tête de l'arc. Dans le cas où l'arête n'est pas dirigée, la communication peut se faire dans les deux sens.

2.1.1 Paramètres du modèle

L'envoi d'information entre deux hôtes est nommé un *appel*. Le mode de communication utilisé dans un réseau nous indique quels appels peuvent être faits par un hôte pendant une unité de temps et aussi l'information qui peut être échangée durant cet appel. Par exemple, dans le cas des communications radios, un hôte diffuse l'information à tous ceux qui sont dans son rayon de transmission. Autrement dit, un noeud envoie un message à tous ses voisins dans une même phase. On nomme ce mode de communication *n-port*. Si on prend le cas des communications réseaux cablés, un hôte doit envoyer l'information sur un lien qui le relie à un autre hôte du réseau. Souvent, l'information est envoyée à un seul voisin par phase, c'est le mode de communication *1-port*.

Dans chacun de ces deux cas, *n-port* et *1-port*, on doit définir aussi l'information pouvant être échangée lors de ces messages. Lorsque les messages ne peuvent aller que dans une seule direction à la fois sur les arêtes du graphe qui représente le réseau, on dit que l'on est dans le modèle *bidirectionnel à l'alternat*. Ceci revient à dire que lorsque deux hôtes communiquent, seulement l'un d'entre eux peut envoyer un message à l'autre

au cours d'une phase. Nous pouvons aussi être dans le cas d'un modèle moins restrictif, où les messages peuvent aller dans les deux directions en même temps durant une phase. Lorsque deux hôtes communiquent, ils peuvent s'envoyer mutuellement des messages. Ce modèle est appelé *bidirectionnel simultané*.

Le modèle des pannes doit décrire quelles composantes du réseau peuvent être affectés par les pannes. Nous devons définir si ce sont les hôtes ou les liens qui peuvent être affectés par les pannes. Il peut aussi arriver que les deux types de composantes soient affectées à la fois. Le modèle doit ensuite définir le type de pannes qui sera traîté. Nous subdivisons les pannes en deux types principaux : les pannes *arrêts* et les pannes *byzantines*.

Lorsqu'on est en présence de pannes de type arrêt, soit l'hôte ne peut pas recevoir ou envoyer de messages, soit le lien ne peut plus les transmettre. Ce type de pannes est facile à traiter comparativement aux pannes byzantines. Malgré le fait que certaines informations puissent être perdues, lorsqu'on reçoit un message on peut être certain de la validité de son contenu.

Dans le cas des pannes byzantines, la composante affectée peut agir de manière complètement arbitraire et, certaines fois même, de manière malicieuse. La panne peut ou bien bloquer le message, ou bien l'envoyer à un mauvais endroit, ou bien encore altérer de manière nuisible tout message qui transite par cette composante. Ces pannes peuvent être causées, entre autres, par des agents malicieux dont le but est de nuire au processus de communication. Les pannes byzantines représentent le pire type de scénario pour la communication dans les réseaux. Les algorithmes qui fonctionnent dans le cas des pannes byzantines peuvent être utilisés dans le cas de chaque autre type de pannes.

Une autre caractéristique des pannes qui doit être définie est la durée de celles-ci. L'un des scénarios envisagables est que la panne est présente au début de l'exécution de l'algorithme et qu'elle y demeure pour toute la durée de celui-ci. Ce type de panne est appelé panne *permanente*. Une autre possibilité est qu'une panne soit *transitoire*. Dans ce cas, la panne peut survenir à tout moment de l'algorithme et n'est présente que pendant une unité de temps. Lorsqu'un lien est affecté par une panne transitoire, on dit que c'est une *erreur de transmission*. Il est important de spécifier la durée des pannes dans le réseau, puisque, par exemple, si un lien tombe en panne de manière transitoire, un algorithme pourra envoyer de nouveau l'information sur ce lien, ce qui ne serait pas une bonne stratégie dans le cas de pannes permanentes.

Certaines limitations sur le nombre de pannes dans le réseau doivent être imposées pour permettre une communication entre les hôtes. La distribution des pannes dans le réseau peut être vue selon deux modèles différents. Le premier suppose qu'il existe une borne sur le nombre de pannes pouvant être présentes dans le réseau. Dans le modèle *borné*, une limite supérieure k sur le nombre de pannes est imposé et on suppose qu'elles sont situées de la pire manière possible. L'objectif est d'envoyer le message à tous les noeuds n'étant pas en panne, sachant qu'il n'y a pas plus de k pannes dans le réseau. On appelle un tel algorithme *k-tolérant.*

L'autre modèle, dit *probabiliste*, suppose que les pannes surviennent de manière aléatoire et indépendantes l'une de l'autre, avec une probabilité fixe. Dans ce cas, nous ne pouvons être certains de la terminaison de l'algorithme puisque, avec une petite probabilité, toutes les composantes du réseau peuvent tomber en panne et empêcher toute transmission. Alors, un algorithme *presque certain* est recherché, c'est-à-dire que l'algorithme doit fonctionner avec une probabilité de $1 - \frac{1}{n}$ si le nombre de noeuds n est assez grand. Cela implique que la probabilité que l'algorithme fonctionne converge vers 1. Normalement on peut renforcer la borne $1 - \frac{1}{n}$ à $1 - \frac{1}{n^c}$, pour toute constante positive c, avec seulement l'ajout d'un facteur constant sur le coût de l'algorithme.

On dit qu'un algorithme est non-adaptatif lorsqu'il doit déterminer préalablement toutes les actions devant avoir lieu à chaque phase. Pour ce qui est des algorithmes adaptatifs, chaque hôte du réseau peut déterminer ce qu'il fera dans la prochaine phase avec l'information qui lui est disponible. Les hôtes savent si les appels qu'ils ont faits dans les phases précédentes ont fonctionné et peuvent agir en conséquence dans les phases futures. Cependant, seule l'information disponible à l'hôte peut être utilisée pour déterminer ses actions, il n'existe pas de moniteur central qui sait tout ce qui se passe dans le réseau à chaque instant. Les algorithmes adaptatifs requièrent plus de mémoire et plus de puissance de calcul de la part des processeurs, mais peuvent être plus efficaces que les algorithmes non adaptatifs.

2.1.2 Communication tolérante aux pannes

Deux types de communications sont surtout étudiés dans ce domaine : la *diffusion générale* et l'*échange totale.* Dans le cas de la diffusion générale, un hôte dans le réseau possède l'information et doit la faire parvenir à tous les autres hôtes qui ne sont pas

en panne. L'hôte qui possède l'information de départ est appelé la *source* et on suppose qu'il n'est pas affecté par une défaillance. Pour ce qui est de l'échange total, tous les hôtes n'étant pas en panne doivent échanger avec tous les autres hôtes l'information qu'ils possèdent. Cet échange aboutit souvent à une décision que l'on nommera l'*accord* et dont nous reparlerons plus loin.

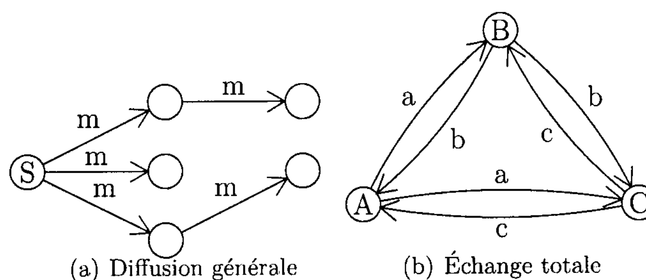


FIG. 2.1 – La diffusion générale et l'échange total

Les deux mesures importantes utilisées pour l'analyse de la performance des algorithmes de diffusion générale et d'échange totale sont le nombre de transmissions élémentaires (*appels*) et le nombre de phases nécessaires à l'algorithme (*temps*). Les concepteurs d'algorithmes doivent souvent faire le choix, pour leurs algorithmes, entre l'efficacité et le degré de tolérance aux pannes. Il est aussi important de prendre en considération la topologie nécessaire à l'accomplissement de ces algorithmes. Puisque les réseaux denses sont très coûteux et difficiles à mettre en opération, il est important de considérer les algorithmes qui fonctionnent sur des réseaux les plus épars possibles, c'est-à-dire dont le degré moyen des noeuds du réseau est le plus faible possible.

Une mesure autre que la performance d'un algorithme de communication est sa robustesse, c'est-à-dire sa capacité de bien fonctionner en présence de pannes d'un certain type avec une certaine distribution. La performance d'un algorithme de communication est souvent en contradiction avec sa robustesse, puisque plus un algorithme de communication est robuste, plus il est lent. La raison de ceci est que la robustesse est acquise grâce à la redondance, qui consiste à envoyer plusieurs fois le même message ou bien à l'envoyer sur différents chemins à la fois. Le principal défi est alors d'avoir la meilleure performance possible qui garantit un certain niveau de robustesse.

Un des problèmes principaux étudiés dans le domaine de communication tolérante aux pannes est celui de l'accord distribué. Nous allons maintenant présenter trois variantes

de ce problème [22] avec les conditions qu'un algorithme doit remplir pour les résoudre. Chacun de ces problèmes suppose que le réseau comporte n processeurs et que chaque processeur reçoit une entrée $v \in V$. À la fin de l'algorithme, chaque processeur doit prendre une décision $v \in V$ satisfaisant trois conditions : l'accord, la validité et la terminaison. Ces trois conditions seront définies de manière différente selon le modèle choisi.

Le premier problème est l'accord probabiliste, en supposant des pannes transitoires probabilistes de type arrêt sur les liens. L'ensemble V ne contient que les valeurs 0 et 1 ($V = 0, 1$). Soit $0 < \epsilon < 1$ un paramètre, appelé *erreur*. L'accord exige que pour chaque choix d'entrées binaires et pour chaque schéma de livraison de message, la probabilité qu'un processeur décide 0 et qu'un autre décide 1 soit au plus ϵ. La validité demande que si toutes les entrées sont 0, alors 0 est la seule décision possible et si toutes les entrées sont 1 et tous les messages sont livrés, alors 1 est la seule décision possible. La terminaison suppose que tous les processeurs prennent la décision. On a montré que l'accord probabiliste est possible pour n'importe quel paramètre ϵ.

Le deuxième et le troisième problèmes, l'accord distribué avec pannes arrêts et l'accord byzantin, supposent un nombre borné de pannes permanentes de noeuds de type arrêt et byzantin, respectivement. Pour les pannes arrêt, l'accord exige que les valeurs décidées par tous les processeurs soient identiques. La validité demande que si toutes les entrées sont égales à v, alors v est la seule décision possible. La terminaison suppose que tous les bons processeurs prennent la décision. Pour l'accord byzantin, la seule chose qui change est que l'exigence concernant les processeurs qui effectuent l'accord est limité aux bons processeurs, puisque nous ne pouvons pas imposer de restriction sur le comportement des processeurs byzantins. Il est prouvé que l'accord avec pannes arrêt est faisable pour n'importe quelle borne f sur le nombre de pannes et l'accord byzantin est faisable si $f < \frac{n}{3}$, où n est le nombre de processeurs.

2.1.3 Diagnostic des pannes

L'objectif du diagnostic des pannes est de déterminer quels sont les processeurs en panne. Par la suite, on peut entreprendre des actions de réparation ou d'échange de processeurs défectueux ou bien tout simplement éviter les composantes en pannes. Nous allons présenter un modèle très connu pour le diagnostic des pannes dans les systèmes

multiprocesseurs : le modèle PMC [18]. Ce modèle utilise la représentation du système de tests par un graphe dirigé où les noeuds sont des processeurs et un arc (u, v) signifie que u peut effectuer un test sur le processeur v.

Dans le modèle PMC, on suppose que les pannes sont permanentes et que leur nombre est borné par un entier f. Chaque test est effectué par un seul processeur et chaque processeur a la capacité de tester tous les autres processeurs, s'ils sont reliés par un arc. Un test fait par un bon processeur peut déterminer avec précision si l'autre processeur est fautif ou non. Les tests faits par un mauvais processeur donnent des résultats quelconques, indépendamment du statut du processeur testé. On considère donc un comportement byzantin des mauvais testeurs. Tous les tests sont récupérés par un observateur central et le diagnostic est ensuite redistribué à tout les processeurs.

Il a été montré que si le système contient f processeurs fautifs alors le système doit contenir n processeurs, où $n \geq 2f + 1$, pour que le diagnostic soit toujours possible. Il faut aussi que chaque processeur soit testé par au moins f autres processeurs.

Il existe aussi d'autres modèles du diagnostic des pannes, comme par exemple le modèle BGM [3] ou bien un modèle basé sur les comparaisons [24], mais ils ne seront pas présentés ici puisque notre objectif est seulement de montrer qu'il existe une manière de détecter les pannes dans le système. C'est une technique qui se rapproche de la détection de trous noirs dans les réseaux, car les trous noirs sont en fait un type de panne.

2.2 L'exploration des graphes par les agents mobiles

Explorer et faire une carte d'un environnement inconnu est un problème qui touche divers domaines, allant de la navigation des robots à la recherche sur le Web. La modélisation du problème varie selon le domaine choisi. Nous allons faire un bref survol de différents problèmes rencontrés lorsque la modélisation est faite à l'aide d'un graphe. Il existe aussi une branche d'exploration par les agents mobiles qui touche aux surfaces géométriques [29, 2], où les agents se déplacent dans une pièce dans laquelle il y a des obstacles et ces agents utilisent soit la vision, soit le toucher pour s'orienter dans un tel environnement. Cependant, ces modèles s'éloignent trop du sujet de ce mémoire et ne seront pas abordés.

Différentes contraintes sur les graphes et les agents sont envisagées. Par exemple les graphes peuvent être dirigés ou non, leurs noeuds peuvent avoir des étiquettes ou ne pas en avoir, les agents peuvent, ou non, avoir une contrainte sur la distance maximale à parcourir avant de revenir au point de départ, etc. Les différents modèles utilisés pour l'exploration à l'aide d'agents ont la caractéristique suivante en commun : les agents commencent sur un noeud particulier qui est le point de départ s et doivent parcourir toutes les arêtes du graphe et, possiblement, faire une carte de ce graphe. Ce graphe doit être connexe pour que les agents puissent parcourir toutes les arêtes. Dans la majorité des cas, un seul agent est nécessaire, c'est seulement lorsque plus d'un est nécessaire que l'on en fera mention.

Le coût calculé pour analyser l'efficacité des algorithmes est le nombre total d'arêtes traversées. Pour ce faire, on considère tous les noeuds du graphe comme points de départ et on prend la plus longue des exécutions en pire cas. Lorsqu'un agent se retouve dans un noeud, il peut choisir d'emprunter soit une arête qu'il a déjà parcourue, soit une arête inconnue. Dans le deuxième cas, c'est un adversaire qui détermine celle qu'il emprunte, car du point de vue de l'agent toutes les arêtes inconnues ont l'air pareilles. Le coût d'un algorithme est donc le pire cas par rapport au nombre d'arêtes parcourues, pris sur tous les points de départ et sur toutes les décisions de l'adversaire.

On peut ensuite comparer l'efficacité de l'algorithme par rapport à l'algorithme de parcours optimal qui connaît la topologie du graphe. Pour faire cette comparaison, on prend le temps d'exécution de l'algorithme trouvé, divisé par le temps d'exécution de l'algorithme de parcours optimal pour ce graphe (algorithme qui traverse toutes les arêtes du graphe au moins une fois). Ensuite on trouve le surplus de l'algorithme donné qui est le maximum de ces rapports pour une famille donnée de graphe. Le surplus mesure la pénalité relative encourue par l'algorithme à cause de l'ignorance de la topologie du graphe. Cette valeur peut être trouvée dans le cas des graphes en général ou bien pour une famille de graphes particulière. Il est intéressant de constater que l'algorithme de parcours optimal dans le cas des graphes eulériens est de passer une seule fois par toutes les arêtes.

2.2.1 Graphes non dirigés avec étiquettes

Le modèle le plus simple est lorsque le graphe n'est pas dirigé et que les noeuds possèdent une étiquette permettant à un agent de les distinguer. L'agent peut commencer

la recherche avec plus ou moins d'information sur la topologie du réseau : différents cas ont été analysés dans [26, 27, 10]. Lorsque l'agent n'a aucune information sur le réseau qu'il va explorer, il a été prouvé [10] que la recherche en profondeur (DFS) est la solution optimale et qu'elle donne un surplus de deux. Aussi, lorsque l'agent connaît la carte du réseau à explorer, mais qu'il ne connaît pas l'emplacement de départ, un algorithme a été trouvé [10] pour la ligne donnant un surplus de $\sqrt{3}$. Il a été prouvé que ce surplus est le plus petit possible. Pour les arbres nous savons que le surplus optimal est strictement inférieur à 2 et qu'il est au moins $\sqrt{3}$. Pour ce qui est des graphes généraux, l'algorithme optimal reste la recherche en profondeur. Si les agents connaissent aussi leur position de départ dans le graphe, le surplus optimal pour la ligne et l'arbre deviennent $\frac{7}{5}$ et $\frac{3}{2}$, respectivement.

Deux indices sont utilisés [27] pour mesurer l'impact du niveau d'information topologique que l'agent possède au départ. L'indice de *sensitivité à l'ignorance de la carte [sic(A, G)]* est la proportion entre le coût d'un algorithme A pour un graphe G ne possédant pas de carte du graphe à explorer et le coût minimal pour G d'un algorithme possédant une carte, mais pas le sens de la direction. De manière similaire, l'indice de *sensitivité à l'ignorance de la direction [sid(A, G)]* est la proportion entre le coût d'un algorithme A pour un graphe G possédant une carte du graphe à explorer, mais pas le sens de la directon et le coût minimal pour G d'un algorithme possédant une carte et le sens de la direction. Pour l'algorithme DFS, la valeur de ces deux indices est de deux pour les graphes généraux. Il a été prouvé que cette valeur est une borne inférieure pour les graphes généraux [27]. Par contre elle peut être inférieure dans le cas de topologies spécifiques.

2.2.2 Graphes dirigés avec étiquettes

Le graphe peut aussi être dirigé, c'est-à-dire que les arêtes possèdent une direction et que les agents ne peuvent parcourir celles-ci que de la queue vers la tête. Il a été noté [1] que le parcours d'un cycle eulérien, si celui-ci existe, est le parcours optimal d'un graphe. La propriété eulérienne d'un graphe, ou bien le degré de rapprochement de celle-ci, est très importante dans ce problème. Débutons par certaines notations : un noeud est appelé un *puit* si son degré d'arc entrant est supérieur à son degré d'arc sortant. Cette différence est appelée la *force* d'un puit. La somme de toutes les forces des puits du graphe est appelée la *carence* d'un graphe. Les graphes eulériens ont une

carence de zéro, puisque la carence est le nombre d'arcs devant être rajoutés au graphe pour le rendre eulérien.

Certaines recherches sur l'efficacité d'exploration sont faites en fonction du coefficient de carence c du graphe. Il a été montré [1] que la borne inférieure de la proportion du coût en pire cas pour un graphe de carence c sur le parcours optimal est de $\Omega(c)$. Sur les graphes de carence 1, un algorithme est proposé pour lequel l'agent ne traverse jamais plus de quatre fois un même arc. Par contre cet algorithme ne peut pas être généralisé pour une carence arbitraire. Un autre algorithme est présenté où l'agent ne traverse jamais plus de huit fois un même arc pour un graphe de carence 1 et qui est généralisé à $2^{O(clogc)}$ pour un graphe de carence c. Cet algorithme n'a pas besoin de connaître c à l'avance.

2.2.3 Graphes sans étiquettes

Lorsque les noeuds ne possèdent pas d'étiquettes uniques servant à les distinguer entre eux, les agents ne peuvent pas savoir s'ils ont déjà rencontré ce noeud par le passé et, sans l'aide de techniques particulières, ne peuvent pas toujours obtenir de carte correcte du graphe. Nous allons voir deux techniques pouvant être utilisées pour produire une carte du graphe : l'utilisation de cailloux [4, 16] pour faire une marque sur un noeud et ainsi l'identifier et une technique simulant les actions d'un automate à états fini déterministe à l'aide de deux agents [5]. Les arêtes d'un noeud sont identifiées de 1 à d et chaque noeud possède d arêtes. Les arêtes d'un noeud doivent être étiquetées pour permettre à un agent de reprendre un chemin déjà parcouru, identifié pendant le fonctionnement de l'algorithme ou bien à l'aide d'une carte du graphe.

Une manière de faire l'exploration d'un tel graphe est l'utilisation de cailloux [4, 16]. Un agent peut laisser un caillou sur un noeud pour pouvoir l'identifier plus tard et ensuite reprendre son caillou. De cette manière l'agent peut utiliser une méthode de retour en arrière dans le cas des graphes qui ne sont pas dirigés et ainsi produire une carte du graphe. C'est la méthode proposée dans [16], qui n'utilise qu'un seul caillou et explore le graphe avec un coût de $O(|V| \cdot |E|)$. Par contre, dans un graphe dirigé, l'utilisation du retour en arrière n'est plus possible. L'agent qui laisse son caillou fait face à l'éventualité de perdre ce dernier et de ne plus pouvoir le retrouver (ou du moins pas avant très longtemps). L'agent qui perd son caillou peut prendre un temps exponentiel en n avant

de le retrouver. Certaines stratégies sont développées pour découvrir un chemin fermé contenant le noeud où le caillou sera déposé, ce qui n'est possible que lorsque l'agent connaît n. Il a été prouvé [4] qu'un agent qui connaît n peut explorer le graphe à l'aide d'un seul caillou, par contre s'il ne connaît pas n, un nombre de $\Theta(\text{loglog}n)$ cailloux est nécessaire et suffisant.

Il est possible qu'un agent ne puisse pas laisser de caillou sur un noeud, dans certaines applications. Malgré cette contrainte supplémentaire, une stratégie est d'utiliser deux agents et de simuler les actions d'un automate à états fini déterministe [5]. Cette stratégie est utilisée dans le cas des graphes dirigés. Pour ce faire, deux agents sont nécessaires et doivent, ou bien pouvoir communiquer entre eux par signal radio, ou bien être synchronisés. Ils doivent utiliser une séquence d'attente et de poursuite. Les deux agents parcourent le même chemin en direction d'un noeud v, mais insèrent des périodes de repos dans ce parcours, faisant en sorte qu'ils ne vont pas à la même vitesse. L'idée est que l'un des agent est à la poursuite de l'autre. À chaque pas, ils notent s'ils sont ensemble ou non et la chaîne ainsi obtenue permet d'identifier le noeud destination selon le noeud de départ. Grâce à cette technique, un algorithme est présenté dans [5] qui permet de construire la carte du graphe avec une très haute probabilité au coût de $O(d^2n^5)$.

2.2.4 Restrictions sur l'agent

Certains modèles [9, 17, 6] imposent des restrictions supplémentaires sur les agents qui explorent le graphe. Une de ces restrictions est l'utilisation d'une laisse qui impose à l'agent une limite sur la distance dont il peut s'éloigner du point de départ et qui le force à revenir par le même chemin au point de départ. Cette laisse doit être au moins de longueur égale à la distance maximale qui sépare le point de départ de tous les autres noeuds du graphe. Autrement, l'agent ne pourrait pas visiter tous les noeuds du graphe. Une autre restriction qui est aussi utilisée est un réservoir d'essence qui impose une limite sur le trajet que l'agent peut parcourir avant de revenir au point de départ pour se réapprovisionner. Pour ce qui est du réservoir d'essence, il doit permettre à l'agent de se rendre au noeud le plus éloigné du point de départ et de revenir.

Ces deux modèles introduisent une contrainte [6] à l'exploration des graphes à l'aide d'agents, l'*interruptibilité*. Cette contrainte est qu'un agent doit retourner au point de départ s après avoir traversé p arêtes. Ce nombre de pas est normalement $2(1+\alpha)r$, où r

est la distance du noeud le plus éloigné du point de s et α est une constante non-négative. Il a été prouvé [9] qu'un algorithme que l'on interrompt pour permettre à l'agent d'aller reprendre de l'essence ne rajoute qu'un facteur constant au nombre de traversés d'arêtes.

Même si dans le cas de l'utilisation d'une laisse, l'agent n'a pas à revenir périodiquement à s, il se peut qu'il doive faire un grand retour en arrière pour visiter un noeud adjacent. Il a été prouvé [17] que le problème d'un agent avec un réservoir d'essence de taille $2(1+\beta)r$ peut être réduit au problème d'un agent avec une laisse de $(1+\alpha)r$, où $\beta > \alpha$. Cette réduction entraine seulement une augmentation d'un facteur constant au coût de l'algorithme.

Il est intéressant de prendre note que pour ces deux modèles les algorithmes de recherche conventionnels ne peuvent pas être utilisés. L'algorithme de recherche en largeur (BFS) n'est pas permis puisqu'il suppose que les agents se transportent d'un noeud à l'autre sans passer par des arêtes. Une version modifiée de BFS et utilisable pour ces modèles est présentée dans [9]. L'algorithme de recherche en profondeur (DFS) va à l'encontre de l'interruptibilité ou bien de la longueur maximale de la laisse. Des algorithmes alternatifs sont étudiés [9, 6] et un algorithme au coût linéaire dans le nombre d'arêtes ($\Theta(|E|)$) est présenté dans [17].

Ces deux modèles se prêtent bien à un problème qui diffère de l'exploration des graphes : la chasse au trésor [9, 17]. Un trésor est caché dans le graphe à une distance δ du point de départ et l'agent doit le retrouver. Nous n'avons donc qu'à diminuer la longueur de la laisse ou du réservoir d'essence pour n'explorer que les noeuds à cette distance du point de départ. Cette chasse au trésor se rapproche quelque peu de la recherche de trou noir, sauf qu'un trou noir élimine un agent, ce qui fait en sorte que l'utilisation d'au moins deux agents est nécessaire dans le cas des trous noirs (voir la section 2.3).

2.2.5 Graphes avec panne sur les liens

Il s'agit ici d'un problème qui est au croisement de la tolérance aux pannes et de l'exploration des graphes à l'aide d'agents. Dans ce modèle [25] un ensemble d'arêtes $F \subset E$ sont en panne et ne peuvent donc pas être traversées par l'agent. Lorsque l'agent se rend au noeud adjacent à l'une de ces arêtes, il découvre cette panne. Au départ de l'algorithme, l'agent ne connaît pas le nombre d'arêtes défectueuses ni leur

placement, mais il possède une carte complète du graphe à explorer. La composante C, qui contient le noeud de départ s, c'est toute la région accessible à l'agent à partir de s et en tenant compte de l'ensemble F. L'objectif de l'agent est de parcourir tous les noeuds de l'ensemble C ; l'agent s'arrête lorsque le dernier noeud de C est visité. Le coût d'un tel algorithme A a été comparé avec le coût d'un algorithme d'exploration optimal A' qui connaît la composante C avant de commencer. La proportion entre ces deux coûts, maximisée selon toutes les configurations F, est appelée le *surplus* de l'algorithme.

Un algorithme est parfaitement compétitif s'il a le plus petit surplus parmi tous les algorithmes d'exploration qui fonctionnent avec ce scénario. L'objectif est d'obtenir un petit surplus : ou bien un algorithme parfaitement compétitif, ou bien un algorithme qui fait une approximation très serrée du plus petit surplus. Des résultats [25] ont été obtenus pour la famille des arbres. Pour la ligne, qui est un type spécifique d'arbre, et pour tous les points de départ, un algorithme parfaitement compétitif est trouvé. Son comportement et son surplus dépendent seulement de la distance entre le point de départ et le bout de la ligne le plus proche. Pour la famille des arbres généraux et un point de départ arbitraire, les auteurs présentent un algorithme avec un surplus d'au plus $\frac{13}{10}$ plus grand que le surplus d'un algorithme parfaitement compétitif.

2.3 Problème des trous noirs

Dans les sections précédentes, nous avons fait un bref survol de la tolérance aux pannes dans les réseaux ainsi que de l'exploration à l'aide d'agents mobiles. Cette section traite d'un problème à l'intersection de ces deux disciplines : les agents doivent être protégés des menaces provenant de leur environnement. L'une de ces menaces est un hôte dangereux, c'est-à-dire un processus stationnaire qui réside dans l'un des noeuds du réseau et qui peut endommager les agents qui le visitent. Essentiellement, la première étape est, si possible, d'identifier cet hôte, donc de déterminer et rapporter sa localisation. Ensuite, deux possibilités s'offrent : ou bien les agents évitent ce site fautif ou bien une activité de réparation est entreprise.

Un hôte particulièrement dangereux pour les agents est un *trou noir* qui détruit tous les agents qui le visitent, sans laisser de traces. Ce type de danger n'est pas rare dans les réseaux, une panne non détectée d'un site peut le transformer en trou noir. La tâche des agents est donc de trouver sans ambiguïté la localisation de ce trou noir ; cette tâche

sera appelée *recherche de trou noir* (RTN). Les agents obéissent tous aux mêmes règles (un algorithme) et la tâche est complétée si au moins un des agents survit et connaît l'emplacement du trou noir. La recherche concernant ce problème se concentre sur la localisation d'un seul trou noir dans les réseaux, donc on suppose qu'il y a toujours au maximum un trou noir dans le réseau, sauf lorsque cela est explicitement spécifié. L'efficacité des algorithmes de recherche sera évaluée sur trois aspects différents : le nombre d'agents nécessaires pour découvrir les trous noirs, le temps utilisé en pire cas, ainsi que le coût, c'est-à-dire le nombre de déplacements utilisés.

Au commencement d'un algorithme de recherche de trou noir, les agents peuvent soit se retrouver tous au même endroit, c'est-à-dire qu'ils sont *colocalisés*, soit être à des endroits différents dans le réseau, ils sont, dans ce cas, dits *dispersés*. De plus, dans certains modèles, les agents ont des étiquettes distinctes leur permettant de s'identifier. Lorsqu'ils n'ont pas d'étiquettes, ils sont appelés des agents *anonymes*.

Il a été noté dans [12] que deux agents au minimum sont nécessaires pour la localisation d'un trou noir. En effet, dans le cas où un seul agent est utilisé, s'il rencontre le trou noir pendant sa recherche, il sera éliminé avant de pouvoir transmettre l'information sur la localisation de ce dernier. De plus, des agents colocalisés et anonymes agissent collectivement comme un seul agent. On peut en déduire qu'il est impossible de trouver un trou noir dans le réseau lorsque les agents commencent l'algorithme au même endroit et qu'ils n'ont pas d'étiquettes distinctes, puisque ce modèle est équivalent à avoir un seul agent.

La recherche de trou noir à l'aide d'agents mobiles est étudiée présentement selon deux modèles différents. Le premier modèle considère que les agents sont *asynchrones*, c'est-à-dire que nous savons que leurs actions prennent un temps fini, mais dont la durée n'est pas bornée. L'autre modèle est le cas où les agents sont partiellement *synchrones* : nous ne savons pas exactement la durée de leurs actions, mais une borne supérieure sur cette durée est connue. Ces deux modèles diffèrent sur plusieurs points : la communication entre agents, l'information nécessaire, ainsi que pour la classe des réseaux pour lesquels la recherche est faisable. Nous allons décrire chacun de ces deux modèles ainsi que les différents résultats obtenus jusqu'à présent.

2.3.1 Modèle asynchrone

La nature du modèle asynchrone impose certaines limites sur les paramètres du problème de recherche de trou noir. En effet, puisque les actions des agents ont un temps fini, mais dont la durée nous est inconnue, certaines constatations peuvent être tirées à prime abord, ces constatations ont été faites la première fois dans [12].

Il nous est impossible de faire la distinction entre un lien lent et un trou noir. L'algorithme utilisé par les agents ne peut donc pas se terminer de manière explicite lorsque nous n'avons pas la certitude du nombre de trous noirs dans le réseau. Il est donc impossible, par exemple, de vérifier s'il y a un trou noir ou pas dans le réseau. C'est pour cela que dans le modèle asynchrone, on fixe le nombre de trous noirs à un seul et on assume que ce nombre est connu des agents. Soit n le nombre de noeuds dans le réseau. Les algorithmes RTN pour le modèle asynchrone doivent donc toujours explorer $n - 1$ noeuds du réseau avant de savoir où se trouve le trou noir. Les agents ne peuvent donc pas savoir avec certitude où est le trou noir s'ils n'ont pas la connaissance de n dans les réseaux généraux.

Cette constatation impose une méthode de communication entre les agents. En effet, si nous n'avons que deux agents et que le premier tombe immédiatement dans un trou noir dès sa première exploration, l'autre ne peut pas savoir avec certitude si le premier est seulement lent ou bien s'il est dans un trou noir. On utilise donc des tableaux blancs qui se retrouvent dans chaque noeud du réseau pour la communication entre les agents. L'accès à ces tableaux se fait de manière juste en exclusion mutuelle.

De part la nature asynchrone des agents, une borne sur le temps réel de l'exécution de l'algorithme n'est pas connue. Deux manières de calculer le temps sont alors employées : le temps idéal, où chaque déplacement des agents est supposé comme étant synchrone et prenant une unité de temps et le délai borné, pour lequel un déplacement nécessite au plus une unité de temps. Lorsque cela n'est pas spécifié, le temps idéal est utilisé.

Une autre observation, qui a été faite dans [13], est que si le réseau n'est pas biconnexe, le problème RTN est impossible à résoudre pour le modèle asynchrone. En effet, si un agent passe par le lien qui mène à un point d'articulation du graphe et si ce lien est très lent, l'autre agent ne pourra pas savoir avec certitude si ce point d'articulation est un trou noir ou bien simplement le lien est lent. Il en résulte que nous ne pouvons pas résoudre le problème RTN pour les réseaux de type ligne ou arbre par exemple.

Une technique qui est fréquemment utilisée pour la recherche de trou noir dans les réseaux asynchrones est la *marche prudente* [13, 11, 15]. Cette technique permet aux agents d'explorer une partie du réseau tout en laissant savoir aux autres agents qu'un port n'est pas dangereux, dès que cette information est disponible. Au départ, tous les ports du réseau sont considérés comme inexplorés. Lorsqu'un agent traverse l'un de ces ports, il indique dans le noeud de départ que ce port est dangereux. Si le noeud destination n'est pas un trou noir, l'agent revient vers le noeud de départ et indique que ce port est sécuritaire. Les agents n'empruntent jamais un port dangereux, donc deux agents ne peuvent pas être détruits par le même noeud. Cette technique permet de diminuer le nombre d'agent nécessaires pour le problème RTN et n'utilise qu'un temps supplémentaire de $O(n)$. La marche prudente a été présentée pour la première fois dans [12] et les auteurs affirment que tous les algorithmes asynchrones n'utilisant que deux agents se doivent d'utiliser la marche prudente.

Selon le modèle utilisé, les agents peuvent avoir plus ou moins de connaissances topologiques sur le réseau qu'ils explorent. Le niveau de connaissance le plus bas est lorsque les agents n'ont aucune connaissance topologique sur le réseau, c'est-à-dire qu'aucun des noeuds n'a d'étiquette, les agents ne savent pas quel est le point d'arrivée des liens incidents à un noeud et ils n'ont pas la carte du réseau où ils se trouvent. Il a été prouvé dans [13] que le nombre d'agents nécessaires pour le problème RTN avec ce niveau de connaissance est de $\Delta + 1$, où Δ est le degré maximal du graphe représentant le réseau. Le coût pour trouver le trou noir dans les réseaux généraux est de $O(n^2)$ lorsque les agents sont colocalisés et distincts. [13]

Les agents peuvent avoir une information supplémentaire sur le réseau qu'ils explorent : un sens des directions. Les noeuds sont étiquetés et les agents peuvent savoir vers quel noeud un port se dirige. Cette information supplémentaire permet de diminuer le nombre d'agents nécessaires pour localiser un trou noir dans un réseau arbitraire à deux lorsque les agents sont colocalisés et distincts, par contre le coût minimal reste de $O(n^2)$ [13].

Lorsque l'on donne encore plus d'information aux agents, soit des connaissances sur la topologie du réseau à explorer, ont peut diminuer le coût minimal de la recherche de trou noir. Certaines topologies ont été explorés plus en détails [12, 11, 14] tandis que d'autres chercheurs ont tenté de découvrir un algorithme général lorsque les agents ont une carte de la topologie du réseau [13, 15].

Dans ce dernier modèle et lorsque les agents sont colocalisés, le nombre d'agents nécessaires reste de deux, le coût est de $O(n + d\log d)$ [15], où d est le diamètre du réseau, ce qui ne contredit pas le coût minimal de $\Omega(n\log n)$ en pire cas [13], mais qui permet d'atteindre un coût de $\Theta(n)$ pour beaucoup de réseaux fréquemment utilisés (ceux qui ont un diamètre de $O(n/\log n)$).

Lorsqu'on se retrouve dans une topologie spécifique, le coût minimal peut être encore plus bas que pour un réseau général. Par exemple l'anneau a été étudié en détail dans [12]. Lorsque les agents sont colocalisés, le nombre d'agents nécessaire est de deux et ils peuvent trouver le trou noir en utilisant $2n\log n + O(n)$ déplacements, avec un temps de $2n\log n + O(n)$, donc le coût et le temps sont de l'ordre de $O(n\log n)$. Il est intéressant de noter que, puisque les agents sont colocalisés, ils peuvent s'entendre sur l'orientation de l'anneau (c'est-à-dire, le sens gauche-droite ou droite-gauche des ports dans un noeud de l'anneau). Le temps peut être diminué, si on utilise plus de deux agents, jusqu'à un temps minimal de $2n - 4$ pour $n - 1$ agents. Lorsque les agents sont dispersés et que l'anneau n'est pas orienté, nous avons besoin d'au moins trois agents et le coût minimal est de $\Omega(n\log n)$ en pire cas lorsque le nombre d'agents est inconnu et de $\Omega(n\log(n-k))$ lorsque l'on sait que k agents sont utilisés. Pour des agents dispersés et un anneau orienté, le trou noir peut être trouvé à l'aide de $k \geq 2$ agents pour un coût et un temps de $O(n\log n)$. Par contre un algorithme présenté dans [14] diminue le temps à $O(n)$ pour des agents dispersés dans un anneau orienté, mais les auteurs ne spécifient pas le coût d'un tel algorithme.

Il a été démontré que dans les topologies de tore et grille de diamètre $O(n/\log n)$, d'hypercube, CCC, Butterfly et de réseau en étoile, le coût de la recherche de trou noir est de $O(n)$ [15]. Ce résultat est obtenu lorsque les agents sont colocalisés et le nombre d'agents est de seulement deux. De plus, dans ces réseaux, la connaissance de l'emplacement de départ ainsi que de la taille du réseau peuvent être obtenues à l'aide d'une technique d'exploration qui prends un temps de $O(n)$ pour ces topologies [15].

2.3.2 Modèle synchrone

Dans les recherches plus récentes, les réseaux partiellement synchrones sont étudiés pour le problème RTN [7, 8]. Les réseaux totalement asynchrones sont rares en pratique et normalement une borne supérieure sur le temps requis à un agent pour effectuer une

action peut être connue. Il devient alors intéressant d'aborder le problème RTN pour ces réseaux partiellement synchrones. Cette borne supérieure sur le temps peut être alors normalisée à une unité de temps, ce qui permet de calculer le temps des algorithmes en assumant que la traversé d'une arrête prend une unité de temps et que le trou noir soit à la pire localisation (ou bien qu'il soit absent, si c'est pire).

Ce modèle permet d'utiliser un mécanisme de délai d'attente pour localiser un trou noir sur n'importe quel graphe à l'aide de seulement deux agents [8]. La technique utilisée est que l'un des agents attend à un noeud v que les agents savent sécuritaire, tandis que l'autre agent explore un noeud adjacent à v. Si après deux unités de temps l'agent qui explore n'est pas de retour, l'autre sait que le trou noir se situe dans le noeud visité, sinon les deux agents savent que ce noeud est sécuritaire et peuvent s'y rendre. Pour un graphe quelconque il suffit de trouver un arbre couvrant tous les noeuds et de faire une recherche en profondeur utilisant cette technique. Il est intéressant de noter que cette technique est une adaptation de la marche prudente. Le défi dans les réseaux partiellement synchrones n'est donc pas de prouver qu'un algorithme existe pour résoudre le problème RTN, mais de trouver des algorithmes améliorant le temps. Nous devons faire la distinction entre le temps que prend l'exécution de l'algorithme RTN, c'est-à-dire le nombre maximal de déplacements effectués par les agents, et le temps de prétraitement : le temps nécessaire pour trouver quel sera le schéma d'exploration utilisé par les agents.

L'utilisation de tableaux blancs pour l'échange d'informations entre les agents n'est plus nécessaire dans le modèle partiellement synchrone. Les agents peuvent maintenant se rencontrer et même si la rencontre n'a pas lieu après un temps fixé, une information peut en être déduite sur la localisation du trou noir. Dans les articles utilisant le modèle partiellement synchrone [7, 8], les agents sont colocalisés, ils connaissent la topologie du réseau à explorer et un seul trou noir est présent dans le réseau.

Certaines topologies spécifiques ont été analysées. Un algorithme optimal a été trouvé pour la ligne et est présenté dans [7]. Cet algorithme travaille en temps linéaire et prend un temps de $4n - 2$ en pire cas. Dans le même article, la recherche du trou noir dans les arbres est analysée et un algorithme optimal est présenté pour une famille très spécifique d'arbres. Les auteurs présentent aussi un algorithme approximatif pour la famille générale des arbres.

Considérons la famille τ des arbres enracinés ayant la propriété suivante : chaque noeud interne d'un arbre dans τ (incluant la racine) a au moins deux enfants. Le nombre

d'arêtes de l'arbre menant à une feuille est dénoté par γ et le nombre d'autres arêtes est dénoté par ρ. Pour cette famille d'arbres, un algorithme optimal est présenté et fonctionne en un temps $3\rho + \gamma$ lorsque le nombre d'arrêtes est pair et $3\rho + \gamma + 1$ lorsque ce nombre est impair. Pour les arbres en général, un algorithme qui atteint un rapport d'approximation de 5/3 a été trouvé. Pour ces deux topologies de réseaux, la ligne et l'arbre, le problème RTN ne pouvait pas être résolu dans le cas du modèle asynchrone, d'où l'intérêt de ces résultats.

Dans le cas des réseaux généraux, un temps de pré-traitement doit être effectué par les agents avant de commencer l'exploration. Puisque la topologie du réseau n'est pas connue à l'avance, le schéma d'exploration doit être décidé lorsque l'on reçoit la carte du réseau. Il a été prouvé dans [8] que cette recherche d'un schéma optimal est un problème NP-complet car le problème RTN se réduit au problème de recherche de cycle hamiltonien dans les graphes. Un algorithme avec un rapport d'approximation de 9.3 est tout de même présenté dans cet article.

Plusieurs questions restent ouvertes pour le problème RTN : d'autres topologies fréquemment utilisées, l'augmentation du nombre de trous noirs, analyse du cas d'agents dispersés, diminution du ratio approximatif de 9.3 pour les réseaux généraux, etc.

Chapitre 3

Principes méthodologiques et propriétés de base

Ce chapitre a pour objectif de présenter les propriétés de base rencontrées dans ce mémoire, ainsi que la méthodologie utilisée. Nous présentons les principes méthodologiques du mémoire, le modèle utilisé, ainsi que certains résultats pour les graphes en général.

3.1 Méthodologie

La topologie du tore a été choisie à cause de sa simplicité et de sa popularité dans les applications. Nous avons débuté notre analyse par les anneaux qui sont une catégorie très simple de tore mais très populaire en pratique. Ensuite, nous avons abordé le tore de grandeur 2 par k puisqu'il est une catégorie spéciale de tore. Finalement nous avons trouvé la borne inférieure sur les tores en général, pour s'apercevoir qu'une subdivision du tore en sous-tores était une stratégie très efficace. C'est pour cette raison que nous avons aussi un algorithme pour le tore 3 par k.

Sur toutes ces différentes catégories de tore, nous avons fait l'analyse du temps minimal théorique de recherche de trou noir pour les différentes topologies étudiées. Ce temps minimal théorique est appelé borne inférieure. Nous avons commencé par trouver des résultats pour les graphes généraux. Ensuite, nous avons utilisé plusieurs techniques, dont l'induction, pour trouver la borne inférieure.

Ensuite, nous avons conçu des algorithmes, c'est-à-dire des procédures déterministes, qui nous ont permis de résoudre le problème de recherche de trou noir dans les réseaux informatiques ayant une topologie de tore. Nous avons calculé la complexité de ces algorithmes afin de les comparer avec la borne inférieure.

Nous avons déterminé que certains de ces algorithmes ont un temps de fonctionnement égal à la borne inférieure. Lorsque nous n'avons pas pu trouver un tel algorithme, nous avons trouvé un algorithme d'approximation avec taux 1.3 qui est asymptotiquement optimal dans certains cas. Un algorithme est asymptotiquement optimal lorsque le rapport du temps de fonctionnement à la borne inférieure converge vers 1 lorsque le nombre de noeuds du réseau augmente. Un algorithme d'approximation à taux α a le temps de fonctionnement au plus α fois plus grand que la borne inférieure.

Finalement, nous avons produit un logiciel qui fait l'analyse de l'exploration des petits anneaux. Ce programme se trouve à l'annexe A.

3.2 Modèle et terminologie

On considère un réseau $R = \{V, E, s\}$, où V est l'ensemble des noeuds, E est l'ensemble des arêtes et $s \in V$ le point de départ des deux agents. Le noeud s est supposé sécuritaire (s n'est pas un trou noir), les agents sont synchrones, c'est-à-dire qu'il existe une borne supérieure sur le temps utilisé pour traverser une arête et les agents connaissent cette borne. Ils ont des étiquettes distinctes et on suppose qu'il y a au plus un trou noir dans le graphe. Un trou noir est un noeud qui détruit tout agent le visitant. Un algorithme de recherche de trou noir (*RTN*) pour l'entrée R est une séquence de traversées d'arêtes (déplacements) ayant les propriétés suivantes pour chacun des deux agents :

- La borne supérieure sur la traversée d'arête est normalisée à 1, on considère donc que chaque déplacement prend une unité de temps.
- Une fois l'algorithme complété, au moins l'un des deux agents, n'ayant pas visité de trou noir, a survécu et, si le réseau contient un trou noir, cet agent en connaît la localisation. Si le réseau ne contient pas de trou noir, l'agent le sait.

Le temps d'exécution d'un algorithme RTN pour une entrée R est le nombre d'unités de temps qui se sont écoulées jusqu'à la terminaison de l'algorithme, en assumant que le trou noir soit à la pire localisation (ou bien qu'il soit absent, si c'est pire). Un algorithme

est considéré optimal pour une entrée donnée si son temps d'exécution est le plus court possible pour cette entrée.

Pour tous les noeuds du réseau on définit les deux états suivants :

- *Noeud inconnu* : l'un des agents survivants ne sait pas si un trou noir y est situé
- *Noeud connu* : tous les agents survivants savent si un trou noir y est situé

Le *territoire connu* au temps t est composé de l'ensemble des noeuds connus à cet instant. Au début, le territoire connu est uniquement composé du noeud de départ s. On dit qu'un *rendez-vous* survient au noeud v à un temps t lorsque les agents se rencontrent au noeud v et échangent de l'information pertinente, c'est-à-dire qui *augmente strictement* le territoire connu. Le noeud v est appelé un *point de rendez-vous*.

Durant chaque pas de l'algorithme, un agent peut soit traverser une arête, soit attendre sur un noeud. Les deux agents peuvent aussi se rencontrer. Si au temps t un rendez-vous se produit, alors le territoire connu au temps t est défini comme le territoire connu *après* l'échange d'informations entre les agents. La séquence de pas entre deux rendez-vous est appelée une *phase*.

Tout algorithme RTN doit avoir la propriété suivante : après un nombre fini de phases, au moins un agent survit et tous les noeuds sont connus (il y a au plus un trou noir, alors lorsque le trou noir est trouvé, tous les noeuds sont marqués comme connus).

3.3 Résultats généraux pour les graphes

L'objectif de cette section est de trouver les restrictions sur le comportement des agents au cours d'une phase et les procédures de bases pour l'exploration du réseau par les agents. Les lemmes suivants sont une modification des lemmes que l'on retrouve dans [7].

LEMME 3.1. *Durant une phase p, les deux agents ne peuvent pas visiter un même noeud inconnu n*

Démonstration : Soit v un noeud inconnu à une phase p. Si un trou noir se retrouve à cet endroit et que les deux agents visitent v, ils seront tous deux détruits, l'algorithme ne serait donc pas un algorithme-RTN. □

LEMME 3.2. *Un agent ne peut visiter plus d'un noeud inconnu durant une phase p.*

Démonstration : Si un agent visite plus d'un noeud inconnu et que l'un de ces noeuds est un trou noir, il sera détruit avant d'indiquer au deuxième agent que les autres noeuds ne sont pas des trous noirs. L'autre agent n'aura pas le choix de visiter lui-même tous les noeuds visités par le premier agent pour déterminer lequel est le trou noir et sera détruit lui aussi. L'algorithme ne serait donc pas un algorithme-RTN. □

Un noeud inconnu ne peut donc être exploré que lorsqu'il est relié par une arête à un noeud du territoire connu.

LEMME 3.3. *Durant une phase, le territoire connu augmente de un ou deux noeuds.*

Démonstration : D'après le lemme 3.2, un agent ne peut explorer plus d'un noeud inconnu par phase et on n'a que deux agents. Donc, deux nouveaux noeuds au plus viendront s'ajouter au territoire connu à la fin d'une phase. Par ailleurs, la définition d'une phase impose que le territoire connu doit augmenter avant la phase suivante. Le territoire connu sera donc augmenté au minimum d'un nouveau noeud. □

Une *1-phase* est définie comme une phase menant à la découverte d'un seul nouveau noeud. De la même manière, on définit une *2-phase* comme étant une phase menant à la découverte de deux nouveaux noeuds. Le lemme 3.3 démontre que chaque phase est soit une 1-phase soit une 2-phase.

LEMME 3.4. *Supposons que v est le noeud où se produit un rendez-vous après une phase p, alors v fait partie du territoire connu après la phase $p-1$.*

Démonstration : Si v ne fait pas partie du territoire connu, v est donc un noeud inconnu pendant la phase p et selon le lemme 3.1, les deux agents ne peuvent pas s'y retrouver durant la phase p.

Un agent visitant un noeud inconnu doit donc retourner sur le territoire connu pour pouvoir se rendre au noeud de rendez-vous. Les noeuds visités seront donc reliés au territoire connu par une arête au minimum. □

LEMME 3.5. *Le territoire connu est toujours connexe.*

Démonstration : Lorsque nous n'avons qu'un seul noeud, c'est-à-dire le point de départ s, le territoire connu est connexe puisqu'il n'y a qu'un seul noeud. Supposons par induction qu'à la phase p, le territoire connu est connexe. Lors de la phase $p+1$ un ou deux noeuds seront ajoutés au territoire connu (selon le lemme 3.3). Les agents sont sur le territoire connu au début de la phase $p+1$ (lemme 3.4), puisqu'un agent ne peut explorer qu'un noeud inconnu par phase (lemme 3.2), chacun de ces noeuds doit être relié au territoire connu par au moins une arête (sinon l'agent qui l'explore passerait par un autre noeud inconnu), alors le nouveau territoire connu sera donc connexe lui aussi. □

Un noeud u est dit la limite du territoire connu à la phase p si une de ses arêtes incidentes est reliée à un noeud inconnu.

Une manière d'explorer exactement un noeud dans une phase est de faire marcher les deux agents à travers le territoire connu jusqu'à sa limite u. L'un des deux agents visite alors un noeud inconnu et revient rencontrer l'agent l'ayant attendu en u. Si l'on suppose que les deux agents sont à la limite u à un instant t et que (u, v) est l'arête menant au noeud inconnu v, on définit la procédure suivante :

sonder(v) **:** Un agent traverse l'arête (u, v) (qui est vers le noeud v) et retourne au noeud u pour rejoindre l'autre agent qui l'y attend. S'ils ne se rencontrent pas au temps $t+2$, alors le trou noir a été trouvé.

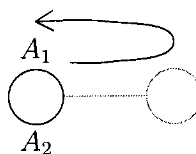


FIG. 3.1 – La procédure *sonder*

On définit aussi une procédure permettant aux deux agents d'explorer deux noeuds inconnus dans une phase. Supposons que les deux agents se retrouvent au noeud u à l'instant t. Soient $v_1, ..., v_i$ les limites du territoire connu à cette phase. Tous les noeuds inconnus pouvant être explorés dans cette phase doivent avoir au moins une arête incidente à l'ensemble $\{v_1, ..., v_i\}$. Soient deux noeuds inconnus k et l ayant chacun une arête incidente à une limite du territoire connu (k, v_k) et (l, v_l), $v_l \in \{v_1, ..., v_i\}$ (possiblement $v_k = v_l$) sélectionnés pour l'exploration. On suppose que u appartient à un des chemins

$< k, l >$ les plus courts reliant les noeuds k et l et inclus dans le territoire connu. La procédure *fourche* est définie de la façon suivante :

fourche(k, l) **:** Un agent traverse le chemin du noeud u au noeud k et retourne vers le noeud v_l. L'autre agent traverse le chemin du noeud u au noeud l et retourne vers le noeud v_k. Dans le chemin $< k, l >$ entre k et l, le noeud k porte l'étiquette 0 et les noeuds restants sont numérotés par les nombres naturels consécutifs en ordre croissant. Soit m l'étiquette du noeud u. Nous dénotons par $dist(k, l)$ le nombre d'arêtes sur le chemin du noeud k vers le noeud l. Si les deux agents ne se rencontrent pas au temps $t + dist(k, l)$, au point de rendez-vous $m' = k + l - m$, alors le trou noir a été trouvé.

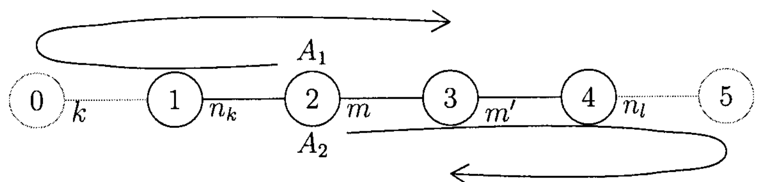


FIG. 3.2 – La procédure *fourche*

On définit aussi la procédure *marcher*, qui consiste à faire un déplacement sur un des chemins les plus courts fixé dans le territoire connu en direction d'un noeud k :

marcher(k) **:** Les deux agents font un pas vers le noeud k.

La procédure *aller* consiste à déplacer les deux agents jusqu'au noeud k.

aller(k) **: Répète** marcher(k) **jusqu'à ce que** les deux agents soient à k

Finalement la procédure *marcher-et-sonder* permet d'explorer tous les noeuds sur le chemin le plus court entre la position courante des agents et le noeud v_k :

marcher-et-sonder(v_k) **:**

Soit $\{v_0, v_1, v_2, ..., v_{k-1}, v_k\}$ un des chemins les plus courts fixé entre le noeud v_0 où les agents se trouvent et v_k,

pour $i = 1$, **tant que** $i \neq k$ **faire**

  si v_i ne fait pas partie du territoire connu **alors**

    aller(v_{i-1}) ;

    sonder(v_i)

Chapitre 4

Résultats pour l'anneau

On considère un anneau $A = \{V, E\}$ où $V = \{0, ..., n-1\}$, $E = \{[i, (i+1)] : i = 0, ..., n-1\}$ et un noeud s qui est le point de départ des deux agents. Toutes les opérations arithmétiques sur les indices sont faites modulo n.

Nous prouvons d'abord une borne inférieure sur le temps de fonctionnement de chaque algorithme RTN travaillant dans un anneau de grandeur n. Ensuite nous présentons un algorithme RTN travaillant en temps égal à cette borne, pour chaque n. Notre algorithme est donc optimal.

LEMME 4.1. *Soient $m \geq 7$ le nombre de noeuds dans le territoire connu, $n \geq 8$ le nombre de noeuds dans l'anneau. Considérons un moment de rendez-vous et soit x la distance minimale séparant les agents de la limite du territoire connu et $i = n - m$ le nombre de noeuds inconnus. Alors le temps minimal requis pour terminer la recherche du trou noir est de $x + 3i - 1$.*

Démonstration : Lorsque $i = 1$, les agents doivent se rendre à l'une des deux limites du territoire connu pour explorer un nouveau noeud. Le temps minimal pour se rendre à ces limites est soit de x unités de temps, soit de $m - 1 - x$ unités de temps. Puisque nous savons que $m \geq 2x + 1$, alors $m - 1 - x \geq 2x + 1 - 1 - x = x$. Ensuite les agents doivent explorer le noeud inconnu, ce qui prend au moins 2 unités de temps, donc $x + 2$ unités de temps au total. Lorsque nous remplaçons $i = 1$ dans $x + 3i - 1$, nous obtenons $x + 2$, donc le lemme est vrai pour $i = 1$.

Lorsque $i = 2$, les agents peuvent explorer les noeuds restants soit en une phase, soit en deux phases. Dans le cas où ils explorent ces noeuds en deux phases, ils doivent se rendre à l'une des limites du territoire connu, ce qui prend au moins x unités de temps. Ensuite, ils explorent le noeud inconnu, ce qui nécessite au moins 2 unités de temps. Les agents doivent alors explorer le noeud inconnu restant lors de la phase suivante, ce qui prend au moins 3 unités de temps. Donc le temps minimal pour explorer deux noeuds inconnus en deux phases est de $x + 5$ unités de temps.

Si les deux agents explorent les deux noeuds inconnus en une seule phase, ils doivent explorer chacun un noeud inconnu aux deux limites du territoire connu et ensuite se rencontrer quelque part dans le territoire connu. Ensemble, ils devront parcourir au moins l'équivalent de deux fois le nombre d'arêtes du territoire connu plus quatre (chacune des arêtes qui lient le territoire connu avec les noeuds à explorer doit être parcourue deux fois). On divise ce temps par le nombre d'agents, ce qui donne $m + 1$, donc le temps d'exploration est d'au moins $m + 1$. Cependant, $m + 1 \geq x + 5$ puisque nous savons que $m \geq 7$ et que $x \leq \lfloor \frac{m-1}{2} \rfloor$. Lorsque nous remplaçons $i = 2$ dans $x + 3i - 1$, nous obtenons $x + 5$, donc le lemme est vrai pour $i = 2$.

Pour prouver que la borne inférieure est vraie lorsque $i > 2$, une induction sur i est utilisée. Pour tout x et tout $j < i$ on suppose que la borne inférieure est vraie. On fixe maintenant x et i, ce qui donne l'état (i, x) de la recherche, et on évalue toutes les actions possibles en une phase à partir de cet état. Puisque les agents peuvent explorer soit un noeud inconnu, soit deux noeuds inconnus dans une même phase (lemme 3.3), nous aurons besoin de savoir le temps nécessaire pour compléter l'exploration à partir des états $(i-1, y)$ et $(i-2, z)$, ce qui donne, par l'hypothèse d'induction, respectivement $y + 3i - 4$ et $z + 3i - 7$, où y est la distance entre les agents et la limite du territoire connu la plus proche pour $i - 1$ et z pour $i - 2$. Nous recherchons le temps nécessaire pour explorer les i noeuds inconnus restants à partir de l'état (i, x).

Si un seul noeud inconnu est exploré dans la phase suivant l'état (i, x), les agents doivent se rendre à l'une des deux limites du territoire connu. Les agents doivent ensuite explorer le nouveau noeud adjacent à cette limite et se rendre à la distance y de l'une des deux limites du nouveau territoire connu, pour atteindre l'état $(i - 1, y)$. Si $y = 0$, cela prend au moins $x + 3$ unités de temps, si $y \geq 1$, ça prend au moins $(x + 2) + (y - 1)$ unités de temps car $y \leq \frac{m-1}{2}$. Le temps de compléter la recherche à partir de l'état $(i - 1, y)$ est au moins de $y + 3i - 4$, donc le temps total est au moins $x + 3 + 3i - 4$ si $y = 0$ et au

moins $(x+2+y-1)+(y+3i-4)$ si $y \geq 1$. Dans chaque cas nous obtenons la borne $x+3i-1$.

Si deux noeuds inconnus sont explorés dans la phase suivant l'état (i,x), les agents doivent chacun explorer un noeud inconnu aux deux limites du territoire connu, se rencontrer quelque part dans ce territoire et se rendre à la distance z de l'une des deux limites du nouveau territoire connu, pour atteindre l'état $(i-2,z)$. Si $z \leq x+1$, cela prend au moins $m+1+x+1-z$ unités de temps, si $z > x+1$, ça prend au moins $m+1+z-(x+1)$ unités de temps. Le temps de compléter la recherche à partir de l'état $(i-2,z)$ est au moins de $z+3i-7$, donc le temps total est au moins $x+m+3i-5$ si $z \leq x+1$ et au moins $2z-x+m+3i-7$ si $z > x+1$. Dans le premier cas nous avons $x+m+3i-5 \geq x+3i-1$ car $m \geq 7$. Dans le second cas, nous pouvons observer que $2z-x+m+3i-7 > x+3i-1$ puisque nous savons que $z > x+1$ et que $m \geq 7$. Par induction, la borne inférieure est donc $x+3i-1$. □

LEMME 4.2. *Soit $n \geq 8$ le nombre de noeuds dans l'anneau, alors le temps minimal requis pour la recherche du trou noir est de $3n-8$.*

Démonstration : Dans cette démonstration, nous dénotons par (m,x) l'état de la recherche au moment de la rencontre, où m est le nombre de noeuds dans le territoire connu et x la distance minimale entre les deux agents et la limite de ce territoire. La programmation dynamique est utilisée pour découvrir le temps minimal pour arriver à chacun des états jusqu'à $m = 8$. Les valeurs dans le tableau représentent le nombre d'unités de temps minimal utilisé jusqu'à présent pour se rendre à l'état $(m(\mathit{ligne}), x(\mathit{colonne}))$. L'algorithme parcourt ensuite les différentes grandeurs de territoire connu m et évalue le temps nécessaire pour conquérir 1 ou 2 noeuds à partir de chaque x pour ce m, puisque les agents ne peuvent explorer que 1 ou 2 noeuds inconnus par phase (lemme 3.3). Une fois ces noeuds découverts, l'algorithme calcule le temps nécessaire à l'atteinte des y distances possibles de la limite du territoire connu.

Pour explorer un noeud inconnu à partir de l'état (m,x) et se rendre à l'état $(m+1,y)$, les agents doivent marcher jusqu'à l'une des deux limites du territoire connu, explorer le noeud inconnu et ensuite se rendre à la distance y de l'une des deux limites du nouveau territoire connu, ce qui prend au moins $x+2+|y-1|$ unités de temps. Donc, le temps calculé pour l'état $(m+1,y)$ est $t+x+|y-1|+2$, où t est le temps utilisé pour se rendre à (m,x).

Pour explorer deux noeuds inconnus à partir de l'état (m, x) et se rendre à l'état $(m+2, y)$, les agents doivent explorer chacun un noeud inconnu aux deux limites du territoire connu, se rencontrer quelque part dans ce territoire et doivent se rendre à la distance y de l'une des deux limites du nouveau territoire connu, ce qui prend au moins $m + 1 + |y - (x+1)|$ unités de temps. Donc, le temps calculé pour l'état $(m+2, y)$ est $t + m + 1 + |y - (x+1)|$, où t est le temps utilisé pour se rendre à (m, x).

Pour présenter les résultats obtenus avec la programmation dynamique, chacun des états est pris en considération et tous les différents états que l'on peut atteindre en une phase sont calculés. La case à la ligne m et colonne x contient le temps minimal courant pour atteindre l'état (m, x). Toutes les cases sauf celle correspondant à $(1, 0)$ sont initialisées à ∞. On représente le résultat du calcul de la manière suivante. Soit $A(m, x)$ l'ancien temps minimal trouvé pour arriver à l'état (m, x) et $N(m, x)$ le nouveau temps minimal trouvé. Si $N(m, x) < A(m, x)$, on **change** l'entrée de la case représentant le temps correspondant à l'état (m, x), sinon on **garde** cette entrée. Si la valeur d'un des états est changée, on affiche le tableau des temps minimaux nécessaire à l'atteinte des différents états, les nouvelles valeurs étant en caractères gras. On conserve le dernier tableau affiché comme étant le tableau contenant les résultats finaux. Les différents tableaux qui suivent sont calculés à l'aide du programme java qui se retrouve dans l'annexe A.

- **Tests pour l'état (1,0)**
  - Ajout d'un noeud supplémentaire : (2,0) = ∞ > 2 **change**
  - Ajout de deux noeuds supplémentaires : (3,0) = ∞ > 3 **change**
  - Ajout de deux noeuds supplémentaires : (3,1) = ∞ > 2 **change**

	0	1	2	3
1	0	-	-	-
2	**2**	-	-	-
3	**3**	**2**	-	-
4	∞	∞	-	-
5	∞	∞	∞	-
6	∞	∞	∞	-
7	∞	∞	∞	∞
8	∞	∞	∞	∞

- **Tests pour l'état (2,0)**
  - Ajout d'un noeud supplémentaire : (3,0) = 3 $\leq$ 5 **garde**

- Ajout d'un noeud supplémentaire : (3,1) = 2 $\leq$ 4 **garde**
- Ajout de deux noeuds supplémentaires : (4,0) = ∞ > 6 **change**
- Ajout de deux noeuds supplémentaires : (4,1) = ∞ > 5 **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	**6**	**5**	-	-
5	∞	∞	∞	-
6	∞	∞	∞	-
7	∞	∞	∞	∞
8	∞	∞	∞	∞

- **Tests pour l'état (3,0)**
  - Ajout d'un noeud supplémentaire : (4,0) = 6 $\leq$ 6 **garde**
  - Ajout d'un noeud supplémentaire : (4,1) = 5 $\leq$ 5 **garde**
  - Ajout de deux noeuds supplémentaires : (5,0) = ∞ > 8 **change**
  - Ajout de deux noeuds supplémentaires : (5,1) = ∞ > 7 **change**
  - Ajout de deux noeuds supplémentaires : (5,2) = ∞ > 8 **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	**8**	**7**	**8**	-
6	∞	∞	∞	-
7	∞	∞	∞	∞
8	∞	∞	∞	∞

- **Tests pour l'état (3,1)**
  - Ajout d'un noeud supplémentaire : (4,0) = 6 $\leq$ 6 **garde**
  - Ajout d'un noeud supplémentaire : (4,1) = 5 $\leq$ 5 **garde**
  - Ajout de deux noeuds supplémentaires : (5,0) = 8 $\leq$ 8 **garde**
  - Ajout de deux noeuds supplémentaires : (5,1) = 7 $\leq$ 7 **garde**
  - Ajout de deux noeuds supplémentaires : (5,2) = 8 > 6 **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	**6**	-
6	∞	∞	∞	-
7	∞	∞	∞	∞
8	∞	∞	∞	∞

- **Tests pour l'état (4,0)**
  - Ajout d'un noeud supplémentaire : (5,0) $= 8 \leq 9$ **garde**
  - Ajout d'un noeud supplémentaire : (5,1) $= 7 \leq 8$ **garde**
  - Ajout d'un noeud supplémentaire : (5,2) $= 6 \leq 9$ **garde**
  - Ajout de deux noeuds supplémentaires : (6,0) $= \infty > 12$ **change**
  - Ajout de deux noeuds supplémentaires : (6,1) $= \infty > 11$ **change**
  - Ajout de deux noeuds supplémentaires : (6,2) $= \infty > 12$ **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	6	-
6	**12**	**11**	**12**	-
7	∞	∞	∞	∞
8	∞	∞	∞	∞

- **Tests pour l'état (4,1)**
  - Ajout d'un noeud supplémentaire : (5,0) $= 8 \leq 9$ **garde**
  - Ajout d'un noeud supplémentaire : (5,1) $= 7 \leq 8$ **garde**
  - Ajout d'un noeud supplémentaire : (5,2) $= 6 \leq 9$ **garde**
  - Ajout de deux noeuds supplémentaires : (6,0) $= 12 \leq 12$ **garde**
  - Ajout de deux noeuds supplémentaires : (6,1) $= 11 \leq 11$ **garde**
  - Ajout de deux noeuds supplémentaires : (6,2) $= 12 > 10$ **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	6	-
6	12	11	**10**	-
7	∞	∞	∞	∞
8	∞	∞	∞	∞

- **Tests pour l'état (5,0)**
  - Ajout d'un noeud supplémentaire : (6,0) $= 12 > 11$ **change**
  - Ajout d'un noeud supplémentaire : (6,1) $= 11 > 10$ **change**
  - Ajout d'un noeud supplémentaire : (6,2) $= 10 \leq 11$ **garde**
  - Ajout de deux noeuds supplémentaires : (7,0) $= \infty > 15$ **change**
  - Ajout de deux noeuds supplémentaires : (7,1) $= \infty > 14$ **change**
  - Ajout de deux noeuds supplémentaires : (7,2) $= \infty > 15$ **change**
  - Ajout de deux noeuds supplémentaires : (7,3) $= \infty > 16$ **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	6	-
6	**11**	**10**	10	-
7	**15**	**14**	**15**	**16**
8	∞	∞	∞	∞

- **Tests pour l'état (5,1)**
  - Ajout d'un noeud supplémentaire : (6,0) $= 11 \leq 11$ **garde**
  - Ajout d'un noeud supplémentaire : (6,1) $= 10 \leq 10$ **garde**
  - Ajout d'un noeud supplémentaire : (6,2) $= 10 \leq 11$ **garde**
  - Ajout de deux noeuds supplémentaires : (7,0) $= 15 \leq 15$ **garde**
  - Ajout de deux noeuds supplémentaires : (7,1) $= 14 \leq 14$ **garde**
  - Ajout de deux noeuds supplémentaires : (7,2) $= 15 > 13$ **change**
  - Ajout de deux noeuds supplémentaires : (7,3) $= 16 > 14$ **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	6	-
6	11	10	10	-
7	15	14	**13**	**14**
8	∞	∞	∞	∞

- **Tests pour l'état (5,2)**
  - Ajout d'un noeud supplémentaire : (6,0) = 11 $\leq$ 11 **garde**
  - Ajout d'un noeud supplémentaire : (6,1) = 10 $\leq$ 10 **garde**
  - Ajout d'un noeud supplémentaire : (6,2) = 10 $\leq$ 11 **garde**
  - Ajout de deux noeuds supplémentaires : (7,0) = 15 $\leq$ 15 **garde**
  - Ajout de deux noeuds supplémentaires : (7,1) = 14 $\leq$ 14 **garde**
  - Ajout de deux noeuds supplémentaires : (7,2) = 13 $\leq$ 13 **garde**
  - Ajout de deux noeuds supplémentaires : (7,3) = 14 > 12 **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	6	-
6	11	10	10	-
7	15	14	13	**12**
8	∞	∞	∞	∞

- **Tests pour l'état (6,0)**
  - Ajout d'un noeud supplémentaire : (7,0) = 15 > 14 **change**
  - Ajout d'un noeud supplémentaire : (7,1) = 14 > 13 **change**
  - Ajout d'un noeud supplémentaire : (7,2) = 13 $\leq$ 14 **garde**
  - Ajout d'un noeud supplémentaire : (7,3) = 12 $\leq$ 15 **garde**
  - Ajout de deux noeuds supplémentaires : (8,0) = ∞ > 19 **change**
  - Ajout de deux noeuds supplémentaires : (8,1) = ∞ > 18 **change**
  - Ajout de deux noeuds supplémentaires : (8,2) = ∞ > 19 **change**
  - Ajout de deux noeuds supplémentaires : (8,3) = ∞ > 20 **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	6	-
6	11	10	10	-
7	**14**	**13**	13	12
8	**19**	**18**	**19**	**20**

- **Tests pour l'état (6,1)**
  - Ajout d'un noeud supplémentaire : (7,0) $= 14 \leq 14$ **garde**
  - Ajout d'un noeud supplémentaire : (7,1) $= 13 \leq 13$ **garde**
  - Ajout d'un noeud supplémentaire : (7,2) $= 13 \leq 14$ **garde**
  - Ajout d'un noeud supplémentaire : (7,3) $= 12 \leq 15$ **garde**
  - Ajout de deux noeuds supplémentaires : (8,0) $= 19 \leq 19$ **garde**
  - Ajout de deux noeuds supplémentaires : (8,1) $= 18 \leq 18$ **garde**
  - Ajout de deux noeuds supplémentaires : (8,2) $= 19 > 17$ **change**
  - Ajout de deux noeuds supplémentaires : (8,3) $= 20 > 18$ **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	6	-
6	11	10	10	-
7	14	13	13	12
8	19	18	**17**	**18**

- **Tests pour l'état (6,2)**
  - Ajout d'un noeud supplémentaire : (7,0) $= 14 \leq 15$ **garde**
  - Ajout d'un noeud supplémentaire : (7,1) $= 13 \leq 14$ **garde**
  - Ajout d'un noeud supplémentaire : (7,2) $= 13 \leq 15$ **garde**
  - Ajout d'un noeud supplémentaire : (7,3) $= 12 \leq 16$ **garde**
  - Ajout de deux noeuds supplémentaires : (8,0) $= 19 \leq 20$ **garde**
  - Ajout de deux noeuds supplémentaires : (8,1) $= 18 \leq 19$ **garde**
  - Ajout de deux noeuds supplémentaires : (8,2) $= 17 \leq 18$ **garde**

- Ajout de deux noeuds supplémentaires : (8,3) = 18 > 17 **change**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	6	-
6	11	10	10	-
7	14	13	13	12
8	19	18	17	**17**

- **Tests pour l'état (7,0)**
  - Ajout d'un noeud supplémentaire : (8,0) = 19 > 17 **change**
  - Ajout d'un noeud supplémentaire : (8,1) = 18 > 16 **change**
  - Ajout d'un noeud supplémentaire : (8,2) = 17 ≤ 17 **garde**
  - Ajout d'un noeud supplémentaire : (8,3) = 17 ≤ 18 **garde**

	0	1	2	3
1	0	-	-	-
2	2	-	-	-
3	3	2	-	-
4	6	5	-	-
5	8	7	6	-
6	11	10	10	-
7	14	13	13	12
8	**17**	**16**	17	17

- **Tests pour l'état (7,1)**
  - Ajout d'un noeud supplémentaire : (8,0) = 17 ≤ 17 **garde**
  - Ajout d'un noeud supplémentaire : (8,1) = 16 ≤ 16 **garde**
  - Ajout d'un noeud supplémentaire : (8,2) = 17 ≤ 17 **garde**
  - Ajout d'un noeud supplémentaire : (8,3) = 17 ≤ 18 **garde**
- **Tests pour l'état (7,2)**
  - Ajout d'un noeud supplémentaire : (8,0) = 17 ≤ 18 **garde**
  - Ajout d'un noeud supplémentaire : (8,1) = 16 ≤ 17 **garde**
  - Ajout d'un noeud supplémentaire : (8,2) = 17 ≤ 18 **garde**
  - Ajout d'un noeud supplémentaire : (8,3) = 17 ≤ 19 **garde**
- **Tests pour l'état (7,3)**

- Ajout d'un noeud supplémentaire : (8,0) $= 17 \leq 18$ **garde**
- Ajout d'un noeud supplémentaire : (8,1) $= 16 \leq 17$ **garde**
- Ajout d'un noeud supplémentaire : (8,2) $= 17 \leq 18$ **garde**
- Ajout d'un noeud supplémentaire : (8,3) $= 17 \leq 19$ **garde**

Puisque le territoire connu augmente de 1 ou 2 noeuds dans une phase (lemme 3.3), la borne inférieure sur le temps pour un anneau de grandeur $n \geq 8$ est le minimum des temps (pris sur toutes les valeurs de x) requis pour atteindre l'état (m, x) pour $m = 7$ ou bien $m = 8$ additionné à $x + 3i - 1$ (lemme 4.1), où $i = n - m$. Les temps en fonction de n sont alors calculés pour ces différents états intermédiaires possibles (tableau 4.1) et on peut conclure que la valeur minimale est de $3n - 8$. □

L'état intermédiaire (m, x)	Temps total en fonction de i	Temps total en fonction de n
$(7, 0)$	$3i + 13$	$3n - 8$
$(7, 1)$	$3i + 13$	$3n - 8$
$(7, 2)$	$3i + 14$	$3n - 7$
$(7, 3)$	$3i + 14$	$3n - 7$
$(8, 0)$	$3i + 16$	$3n - 8$
$(8, 1)$	$3i + 16$	$3n - 8$
$(8, 2)$	$3i + 18$	$3n - 6$
$(8, 3)$	$3i + 19$	$3n - 5$

TAB. 4.1 – Anneau avec territoire connu de 7 ou 8 noeuds

THÉORÈME 4.1. *Le temps requis pour la recherche du trou noir dans un anneau de grandeur n est au moins :*

- *2 unités de temps pour $n = 3$*
- *5 unités de temps pour $n = 4$*
- *6 unités de temps pour $n = 5$*
- *12 unités de temps pour $n = 7$*
- $3n - 8$ *unités de temps, lorsque* $n = 6$ **ou bien** $n \geq 8$

Démonstration : Pour les grandeurs d'anneau $n \leq 7$, nous pouvons prendre toutes les configurations trouvées à l'aide de la programmation dynamique dans le lemme 4.2 et prendre le temps minimal parmi les différentes configurations trouvées pour une grandeur d'anneau donnée. Nous avons donc les temps du tableau 4.2. Pour $n = 3, 4, 5, 7$, nous

avons respectivement 2, 5, 6 et 12 unités de temps. Pour $n = 6$, $3n - 8$ donne bien 10 unités de temps et $3n - 8$ est la borne inférieure pour les anneaux de grandeur $n \geq 8$ (lemme 4.2). □

n	Temps
3	2
4	5
5	6
6	10
7	12

TAB. 4.2 – Temps pour les anneaux $n \leq 7$

Nous présentons maintenant un algorithme dont le temps d'exécution atteint, dans chaque cas, la borne inférieure du théorème 4.1.

ALGORITHME 4.1. *Anneau*

cas où $n = 4$
  sonder($s + 1$) ;
  fourche($s - 1$,$s + 2$) ;
cas où $n = 3, 5, 7$
  pour $i := 1$ **à** $\lfloor \frac{n}{2} \rfloor$ **faire**
    fourche($s - i$,$s + i$) ;
cas où $n = 6$ **ou bien** $n \geq 8$
  fourche($s - 1$,$s + 1$) ;
  fourche($s - 2$,$s + 2$) ;
  marcher(s+1) ;
  marcher-et-sonder($s + n - 3$) ;

THÉORÈME 4.2. *L'algorithme Anneau est optimal.*

Démonstration :

- Lorsque $n = 4$, la procédure *sonder($s + 1$)* prend 2 unités de temps et la procédure *fourche($s - 1$,$s + 2$)* prend 3 unités de temps. Cela donne un total de $2 + 3 = 5$ unités de temps.

- Lorsque $n = 3, 5, 7$ le temps requis pour effectuer toutes les *fourches* est de $\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} 2i$ unités de temps, ce qui donne respectivement 2, 6 et 12 unités de temps.
- Lorsque $n = 6$ ou bien $n \geq 8$, les procédures *fourche(s−1,s+1)* et *fourche(s−2,s+2)* prennent respectivement 2 et 4 unités de temps, la marche prend 1 unité de temps et la procédure *marcher-et-sonder($s + n - 3$)* prend $3(n - 5)$ unités de temps. Cela donne un total de $2 + 4 + 1 + 3(n - 5) - 1 = 3n - 8$ unités de temps.

Selon le théorème 4.1 notre algorithme est optimal. □

Chapitre 5

Résultats pour le tore de grandeur 2 par k

On considère un tore de grandeur 2 par k défini comme suit : $T = \{V, E\}$ où $V = \{x_0, ..., x_{k-1}, y_0, ..., y_{k-1}\}$, $E = \{\{x_i, y_i\} : i = 0, ..., k-1\}\} \cup \{\{x_i, x_{(i+1)}\} : i = 0, ..., k-1\} \cup \{\{y_i, y_{(i+1)}\} : i = 0, ..., k-1\}$ et un noeud s qui est le point de départ des deux agents. Toutes les opérations arithmétiques sur les indices sont faites modulo k.

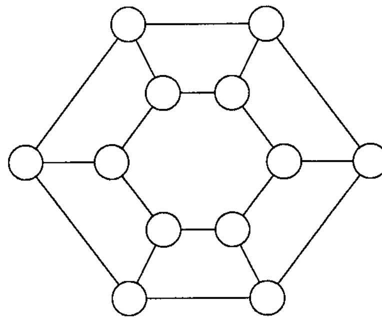


FIG. 5.1 – Tore de grandeur 2 par 6

Nous prouvons d'abord que chaque algorithme RTN dans un tore de grandeur 2 par k utilise au moins le temps de $3k-2$. Ensuite, par une analyse détaillée cas-par-cas, nous prouvons que le temps $3k-2$ n'est pas possible à atteindre ; donc la borne inférieure est en fait $3k-1$. Finalement nous présentons un algorithme RTN travaillant en temps $3k-1$, ce qui est donc optimal.

LEMME 5.1. *L'exploration d'un noeud inconnu coûte au moins deux unités de temps.*

Démonstration : Un agent doit se rendre vers le noeud inconnu, ce qui prend une unité de temps, et selon le lemme 3.4 il doit revenir vers le territoire connu pour la rencontre avec l'autre agent, ce qui prend aussi une unité de temps. □

On définit trois sortes de phases ; les phases légères, normales et lourdes. Une phase est dite *légère* lorsque les deux agents explorent deux noeuds en deux unités de temps. Une phase *normale* peut explorer soit un noeud en deux unités de temps, soit deux noeuds en trois unités de temps. Toutes les autres phases sont dites *lourdes.*

LEMME 5.2. *Pour chaque exploration E dans les trois premières phases, il existe une exploration E' dans les trois premières phases telle que la troisième phase de E' n'est pas légère et le temps, l'ensemble des noeuds explorés et le noeud de rencontre après la troisième phase dans E et E' sont identiques.*

Démonstration : Si la troisième phase est légère, le noeud de départ de cette phase doit avoir deux noeuds adjacents inconnus. Soit un noeud i ayant deux noeuds adjacents inconnus. Ce noeud sera le point de rencontre des agents après la deuxième phase. Dans un tore de grandeur 2 par k les noeuds ont tous le degré trois. Selon le lemme 3.5 le territoire connu doit être connexe. Selon le lemme 3.3 au moins un noeud est exploré par phase, donc le territoire connu de grandeur ≥ 2 au début de la deuxième phase. Alors, il existe un noeud j qui est dans le territoire connu au début de la deuxième phase et qui est adjacent au noeud i.

Puisque la deuxième phase se termine au noeud i, la dernière action que chacun des agent va faire dans l'exploration E est le déplacement du noeud j vers le noeud i. Dans l'exploration E' nous enlevons ce déplacement de j vers i et le nouveau point de rencontre des agents se fait au noeud j. Lors de la troisième phase, nous effectuons le déplacement de j à i et explorons les deux noeuds adjacents à i, ce qui donne une phase normale. Nous avons donc dans E et E' le même temps, le même territoire connu et le même noeud de rencontre i après la troisième phase, mais la troisième phase de E' n'est pas légère. □

Par la suite nous supposons, sans perte de généralité, que la troisième phase d'un algorithme RTN n'est pas légère.

LEMME 5.3. *Si la phase $p > 3$ est légère, alors la phase $p-1$ est lourde.*

Démonstration : Si la phase p est légère, le noeud de départ i de cette phase doit avoir deux noeuds adjacents a et b inconnus après la phase $p-1$. Après la phase $p-1$ le territoire connu doit être de grandeur au moins $(p-1)+1 = p > 3$. Puisque ce territoire doit être connexe, il doit contenir le noeud i, le noeud $j \neq a, b$ adjacent à i et un des deux noeuds $k, l \neq i$ adjacents à j.

Analysons maintenant la phase $p-1$. Puisque les agents doivent se rencontrer au noeud i et les deux noeuds a et b ne seront explorés qu'à la phase p, les agents auront chacun la traversée de l'arête menant de j à i comme la toute dernière action de la phase $p-1$, ce qui prends une unité de temps.

Si les agents n'explorent qu'un seul noeud durant la phase $p-1$, ils utiliseront au moins trois unités de temps, ce qui donne une phase lourde.

Si les agents explorent deux noeuds durant la phase $p-1$, la seule façon de le faire en au plus trois unités de temps avec la rencontre dans i serait de partir de j, explorer les noeuds k et l et se rencontrer dans i. Mais ceci est impossible car pour le faire en au plus trois unités de temps les agents devraient se rencontrer d'abord en j et faire ensemble le trajet à i, ce qui contredit la définition de rencontre en i. Donc, le nombre d'unités de temps nécessaire pour explorer deux noeuds durant la phase $p-1$ est d'au moins quatre unités de temps, ce qui donne aussi une phase lourde. □

LEMME 5.4. *Le temps moyen d'exploration d'un noeud dans une phase normale ou lourde est au moins $\frac{3}{2}$. Le temps moyen d'exploration d'un noeud dans une suite de deux phases, lourde suivie de légère, est au moins $\frac{3}{2}$.*

Démonstration : Par définition une phase normale explore deux noeuds en trois unités de temps et une phase lourde explore deux noeuds en quatre ou plus unités de temps. Une phase normale à donc un temps moyen de $\frac{3}{2}$ et la phase lourde est moins efficace qu'une phase normale, ce qui donne un temps moyen d'exploration d'un noeud dans une phase normale ou lourde d'au moins $\frac{3}{2}$.

Puisqu'une phase lourde prends au moins quatre unités de temps pour deux noeuds explorés et qu'une phase légère prends deux unités de temps pour deux noeuds explorés, la suite de ces deux phases prends au moins six unités de temps pour quatre noeuds explorés. Ce qui donne un temps moyen d'exploration d'un noeud d'au moins $\frac{3}{2}$. □

LEMME 5.5. *Le temps moyen d'exploration d'un noeud dans toutes les phases $p \geq 3$ est au moins $\frac{3}{2}$.*

Démonstration : Considérons la suite des phases $p > 2$, en utilisant le symbole l pour une phase légère, le symbole L pour une phase lourde et le symbole a pour une phase autre que légère. Puisque le premier terme de la suite n'est pas l et chaque l est précédé par L, on peut diviser toute la suite en segments de longueur ≤ 2, ou chaque segment est soit a, soit Ll. En vue du lemme 5.4, le temps moyen d'exploration d'un noeud dans chaque segment est au moins $\frac{3}{2}$, donc le temps moyen d'exploration d'un noeud dans toutes ces phases est aussi au moins $\frac{3}{2}$. La figure 5.2 montre deux exemples d'explorations.

$$* * \lfloor a \rfloor \; \lfloor L\ l \rfloor \; \lfloor a \rfloor \; \lfloor a \rfloor \; \lfloor L\ l \rfloor \ldots$$
$$* * \lfloor L\ l \rfloor \; \lfloor a \rfloor \; \lfloor a \rfloor \; \lfloor L\ l \rfloor \; \lfloor a \rfloor \ldots$$

FIG. 5.2 – Tore 2 par k : Exemples d'explorations

□

THÉORÈME 5.1. *Le temps requis pour un algorithme RTN dans un tore de grandeur 2 par k est au moins de $3k - 2$.*

Démonstration : Lors des deux premières phases, les agents peuvent explorer : soit deux, soit trois, soit quatre noeuds, puisque selon le lemme 3.3, ils doivent explorer au moins un noeud par phase et ils ne peuvent pas en explorer plus de deux. Aussi le temps minimal requis pour ces deux phases est quatre unités de temps pour explorer deux ou trois noeuds et cinq unités de temps pour en explorer quatre.

Selon le lemme 5.5, le temps moyen d'exploration d'un noeud dans toutes les phases $p \geq 3$ est au moins $\frac{3}{2}$. Dans le cas où le nombre de noeuds à explorer après les deux premières phases est pair($2l$), le temps C minimal sera $C \geq 2l \times \frac{3}{2} \geq 3l$. Dans le cas où le nombre de noeuds à explorer après les deux premières phases est impair($2l + 1$), le temps C minimal sera $C \geq (2l + 1) \times \frac{3}{2} \geq 3l + \frac{3}{2}$, mais puisque le nombre d'unités de temps doit être un entier, le temps minimal sera $C \geq 3l + 2$.

Calculons le temps C minimal pour faire l'exploration lorsque l'on explore soit deux, soit trois, soit quatre noeuds dans les deux premières phases. Dans le cas où on explore deux noeuds, il reste $2k - 3$ noeuds à explorer et le temps minimal sera $C \geq 4 + 3(k - 2) + 2 \geq 3k$. Dans le cas où on explore trois noeuds, il reste $2k - 4$ noeuds à explorer

et le temps minimal sera $C \geq 4 + 3(k-2) \geq 3k-2$. Finalement, dans le cas où l'on explore quatre noeuds, il reste $2k-5$ noeuds à explorer et le temps minimal sera $C \geq 5 + 3(k-3) + 2 \geq 3k-2$. □

Nous prouverons maintenant qu'aucun algorithme RTN dans un tore de grandeur 2 par k ne peut fonctionner en temps $3k-2$. Soit $\mathcal{A}$ un algorithme qui travaille en temps $3k-2$. Notre but est de montrer qu'il ne peut pas exister.

LEMME 5.6. *Soit x le nombre de noeuds qui reste à explorer par l'algorithme $\mathcal{A}$ après les deux premières phases. Si x est pair, alors $\mathcal{A}$ explore deux noeuds dans chaque phase $p > 2$. Si x est impair, il existe exactement une phase explorant un noeud et elle prend deux unités de temps.*

Démonstration : D'après la démonstration du théorème 5.1 x peut être soit $2k-4$, auquel cas le temps moyen doit dans toutes les phases $p > 2$ doit être $\frac{3}{2}$, donc chacune doit explorer deux noeuds, soit $2k-5$, auquel cas exactement une phase explore un noeud. □

LEMME 5.7. *Dans l'algorithme $\mathcal{A}$, une phase lourde $p > 2$ doit être suivie par une phase légère.*

Démonstration : Une phase lourde a un temps moyen minimal de deux unités de temps. Pour atteindre le temps moyen de $\frac{3}{2}$, elle doit être combiné avec une phase légère. Puisque selon le lemme 5.3 une phase légère doit être précédé d'une phase lourde, seul ces phases lourde seront acceptées dans l'algorithme $\mathcal{A}$. □

LEMME 5.8. *Dans l'algorithme $\mathcal{A}$, une phase lourde ne peut pas coûter plus de quatre unités de temps.*

Démonstration : Selon le lemme 5.7, les seules phases lourdes pouvant être utilisées sont celles suivies d'une phase légère. Pour atteindre le temps moyen de $\frac{3}{2}$, la phase lourde ne peut pas dépasser quatre unités de temps. □

LEMME 5.9. *L'algorithme $\mathcal{A}$ ne peut pas commencer par l'exploration de trois noeuds lors des deux premières phases.*

Démonstration : Dans le cas où l'on explore trois noeuds dans les deux premières phases en quatre unités de temps, le territoire connu sera sous la forme de la figure 5.3, où R est le point de départ des agents pour la troisième phase. Selon le lemme 5.6 chaque phase $p > 2$ doit explorer deux noeuds, d'après le lemme 5.7 toutes les phases lourdes doivent être suivies d'une phase légère et finalement le lemme 5.8 démontre qu'une phase coûtant plus de quatre unités de temps est impossible.

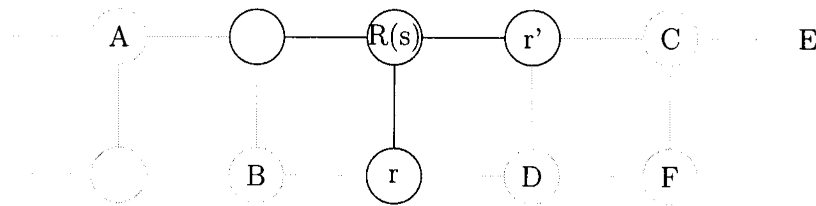


FIG. 5.3 – Tore 2 par k : Deux premières phases

La troisième phase devra explorer deux noeuds parmi les noeuds A, B, C et D. Les combinaisons (A, C), (A, D) et (B, C) ne peuvent pas être effectuées en trois unités de temps, puisque deux déplacements en plus de l'exploration seraient nécessaires. S'ils sont faits en quatre unités de temps, le point de rencontre ne pourrait pas être suivi d'une phase légère. Il ne reste que les combinaisons (A, B), (B, D) et (C, D).

Si les agents explorent les noeuds (B, D), ils vont se rencontrer au noeud r et devront utiliser trois unités de temps. La phase suivante va prendre au moins quatre unités de temps puisque les agents devront faire au moins deux déplacements, soit pour se rendre au noeud à explorer, soit pour se rencontrer, puis ils ne pourront pas se rencontrer à un noeud permettant une phase légère.

Il ne reste que les noeuds (A, B) ou (C, D). Nous n'allons analyser que le couple (C, D), puisque l'analyse du couple (A, B) pourrait être effectué de la même manière par symétrie. Il est impossible de faire une phase lourde suivie d'une phase légère. Si l'exploration se fait en trois unités de temps, le point de rencontre après avoir exploré les noeuds C et D sera r'. Ce qui nous mène à une situation où les seuls noeuds qui peuvent être explorés en trois unités de temps sont les noeuds E et F.

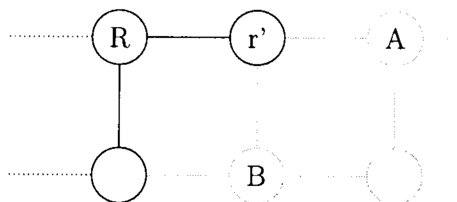


FIG. 5.4 – Tore 2 par k : Configuration d'exploration

Donc chaque phase subséquente débute avec une configuration représentée à la figure 5.4, où R est le point de départ des agents, jusqu'à la phase p, qui est au plus l'avant-dernière phase. Les agents ne peuvent qu'explorer les noeuds A et B et le seul point de rencontre possible est r' après trois unités de temps, ce qui donne une configuration de la figure 5.4 à la phase $p+1$. (Il est à noter qu'on ne peut pas faire une phase lourde suivie d'une phase légère.)

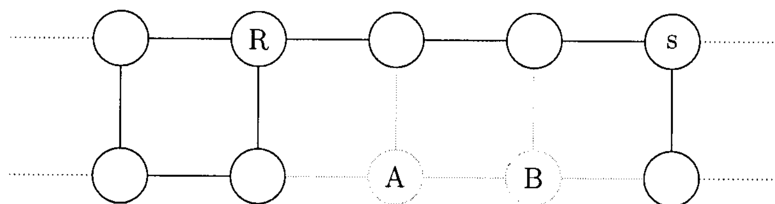


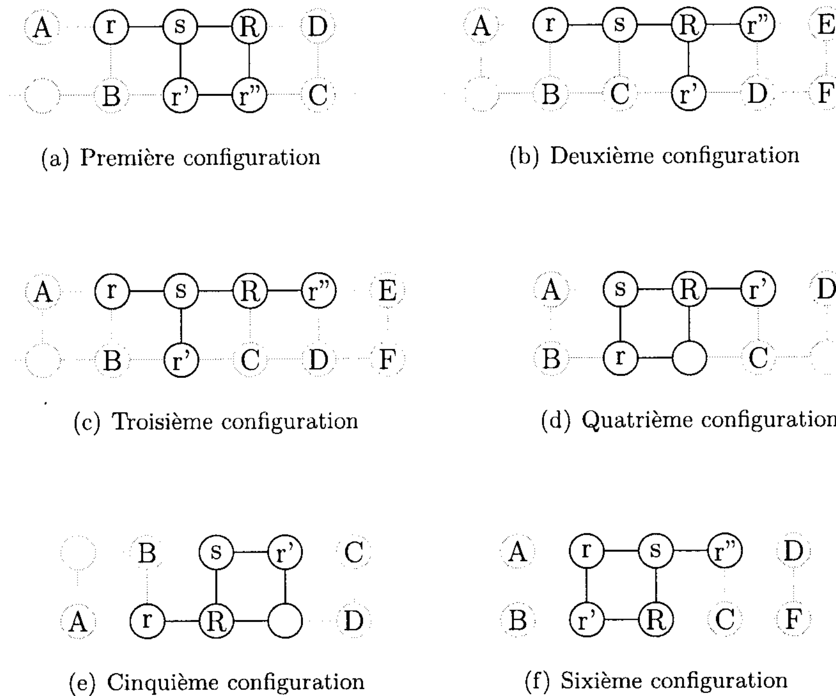
FIG. 5.5 – Tore 2 par k : Dernière phase de l'exploration

Pour la dernière phase, les agents vont se retrouver dans la situation présentée à la figure 5.5, où R est le point de départ de la phase, et où ils doivent explorer les noeuds A et B. Pour explorer le noeud B, un des agents devra faire deux déplacements (en plus de l'exploration), alors la phase prendra au moins quatre unités de temps et ne peut pas être suivie d'une phase légère. Donc, le temps total dans le cas où on explore trois noeuds dans les deux premières phases n'est pas $3k-2$.

□

LEMME 5.10. *L'algorithme $\mathcal{A}$ ne peut pas commencer par l'exploration de quatre noeuds lors des deux premières phases.*

Démonstration : Dans le cas où on explore quatre noeuds dans les deux premières phases en cinq unités de temps, le territoire connu peut avoir douze configurations possibles. La figure 5.6 montre six de ces configurations, où s est le point de départ de l'algorithme et R la localisation des agents pour le début de la troisième phase. Les six autres configurations peuvent être trouvées par symétrie. Selon le lemme 5.6 toutes les phases doivent explorer deux noeuds sauf une qui devra utiliser un maximum de deux unités de temps. Ensuite, le lemme 5.7 indique que toutes les phases lourdes doivent être suivis d'une phase légère et le lemme 5.8 montre qu'une phase coûtant plus de quatre unités de temps est impossible.



(a) Première configuration

(b) Deuxième configuration

(c) Troisième configuration

(d) Quatrième configuration

(e) Cinquième configuration

(f) Sixième configuration

FIG. 5.6 – Tore 2 par k : Configurations possibles

Dans ce qui suit, lorsque l'on dit que les agents doivent faire une exploration en trois unités de temps, c'est qu'ils ne peuvent pas atteindre un noeud permettant une phase légère en utilisant une unité de temps supplémentaire.

Pour la **première configuration**, qui est montrée à la figure 5.6(a), les noeuds A, B, C et D peuvent être explorés en troisième phase. Si les agents explorent le couple (A, B) ou bien (A, D), ils devront utiliser quatre unités de temps et se rencontrer au noeud r, qui ne permet pas de suivre avec une phase légère. Le couple (A, C) est impossible à explorer en moins de cinq unités de temps. Lorsque les agents explorent le couple (B, C) ou (B, D), ils utilisent quatre unités de temps et se rencontrent au noeud r', qui ne permet pas de suivre avec une phase légère. Si les agents explorent le couple (C, D), ils utilisent trois unités de temps et se retrouvent au noeud r''. Suite à cette phase, ils ne peuvent plus faire une exploration de deux noeuds en trois unités de temps ou d'un seul noeud en deux unités de temps. De plus, toutes les autres explorations prenant quatre unités de temps placent les agents sur un noeud où ils ne peuvent pas faire une phase légère.

Il ne reste que l'exploration du noeud D avec une phase explorant un noeud en deux unités de temps. Lors de la phase suivante, nous avons déjà vu que les différentes combinaisons entre les noeuds A, B et C ne sont pas viables. Le nouveau noeud accessible est à une distance de trois arrêtes avec les noeuds A et B, alors un temps d'exploration d'au moins cinq unités de temps. Il ne reste que le couple C avec ce nouveau noeud accessible, ce qui nous place dans la même configuration que l'on peut voir à la figure 5.4. On peut utiliser l'induction décrite dans le lemme 5.9 pour les phases suivantes jusqu'à la dernière. Pour la dernière phase, on se retrouve avec exactement la même configuration qu'à la figure 5.5, mais nous avons démontré dans le lemme 5.9 qu'il était impossible de faire cette exploration en moins de quatre unités de temps. Ce qui nous donne un temps d'au moins $3k - 1$ unités de temps pour cette configuration de départ.

Pour la **deuxième configuration**, qui est montrée à la figure 5.6(b), les noeuds A, B, C, D et E peuvent être explorés en troisième phase. Les couples (A, D), (A, E), (B, D) et (B, E) ne peuvent pas être explorés en moins de cinq unités de temps. Si les agents explorent le couple (A, B), (A, C) ou bien (B, C), ils devront utiliser quatre unités de temps et se rencontrer au noeud r, ce qui ne permet pas de suivre avec une phase légère. De même que le couple (C, E), où la rencontre est au noeud R.

Si les agents explorent le couple (C, D), ils devront le faire en trois unités de temps et se rencontrer au noeud r'. À partir du noeud r', seuls les couples (B, F) et (E, F) peuvent être explorés en moins de cinq unités de temps. L'exploration de (B, F) prend quatre unités de temps et la rencontre se fait au noeud r', qui ne permet pas une phase

légère. S'ils explorent le couple (E, F), ils doivent le faire en trois unités de temps et se rencontrent au noeud r''. Suite à cette phase, ils ne peuvent plus explorer deux noeuds en trois unités de temps et aucune exploration en quatres unités ne permet de suivre avec une phase légère.

Il ne reste alors que le couple (D, E) pour l'exploration en troisième phase de cette deuxième configuration. Suite à cette exploration, on se retrouve dans la même configuration que l'on peut voir à la figure 5.4. On peut utiliser l'induction décrite dans le lemme 5.9, en observant que l'on ne peut pas faire une exploration d'un seul noeud en deux unités de temps durant ces explorations, pour les phases suivantes jusqu'à l'avant-dernière. La figure 5.7 montre la configuration que l'on obtient lors de l'avant-dernière phase. L'exploration du couple (G, B) prend quatre unités de temps et ne peut pas être suivie d'une phase légère.

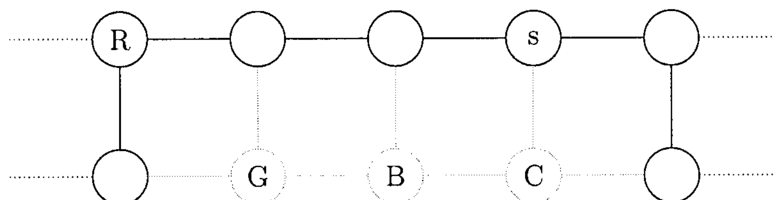


FIG. 5.7 – Tore 2 par k : Deuxième configuration, dernière phase

Pour la **troisième configuration**, qui est montrée à la figure 5.6(c), les noeuds A, B, C, D et E peuvent être explorés en troisième phase. Les couples (A, D), (A, E), (B, D) et (B, E) ne peuvent être explorés en moins de cinq unités de temps. Si les agents explorent le couple (A, B) ou bien (A, C), ils devront utiliser quatre unités de temps et se rencontrer au noeud r, ce qui ne permet pas de suivre avec une phase légère. De même que le couple (B, C), où la rencontre est au noeud r'.

Si les agents explorent le couple (C, D), ils doivent le faire en trois unités de temps et se retrouvent au noeud r''. Deux actions sont ensuite possibles : l'exploration du couple (E, F) avec point de rencontre au noeud D et l'exploration en deux unités de temps du noeud E. S'ils explorent le couple (E, F), ils ne pourront plus explorer de noeud en moins de quatre unités de temps et, dans ce cas, ils ne pourront pas se positionner pour

une phase légère. S'ils explorent le noeud E, ils se retrouvent dans une configuration identique à la figure 5.4. On peut utiliser l'induction décrite dans le lemme 5.9 pour les phases suivantes jusqu'à la dernière. Pour la dernière phase, on se retrouve avec exactement la même configuration qu'à la figure 5.5, mais nous avons montré dans le lemme 5.9 qu'il était impossible de faire cette exploration en moins de quatre unités de temps. Nous obtenons alors un temps d'au moins $3k - 1$.

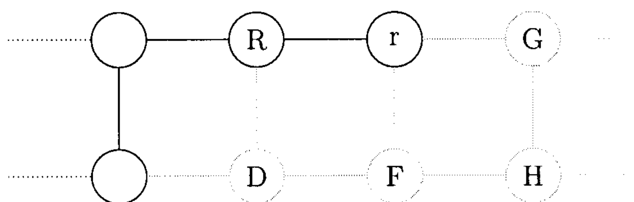


FIG. 5.8 – Tore 2 par k : Induction sur la troisième configuration

Si les agents explorent le couple (C, E), ils doivent le faire en trois unités de temps et se rencontrent au noeud r''. Suite à cette exploration, les agents ne peuvent qu'explorer les noeuds D, F et le nouveau noeud accessible. Ce qui nous donne la configuration montrée à la figure 5.8. Ils ne peuvent faire qu'une exploration de deux noeuds en trois unités de temps ou bien une exploration du noeud D en deux unités de temps. S'ils explorent le couple (F, G), qui doit se faire en trois unités de temps avec le noeud r comme point de rencontre, le noeud D devra être exploré à la phase suivante, sans quoi il sera isolé et ne pourra plus être exploré avec les types de phases admises. Cependant, si les agents explorent le noeud D avec un autre noeud durant la phase suivante, ils devront le faire en quatre unités de temps et ne pourront se positionner pour une phase légère. Si les agents explorent le noeud D avec deux unités de temps, ils se retrouvent dans une configuration identique à la figure 5.4. On peut utiliser l'induction décrite dans le lemme 5.9 pour les phases suivantes jusqu'à la dernière. Pour la dernière phase, on se retrouve avec la même configuration qu'à la figure 5.5, mais nous avons montré dans le lemme 5.9 qu'il est impossible de faire cette exploration en moins de quatre unités de temps. Nous obtenons alors un temps d'au moins $3k - 1$. Une autre option possible est d'explorer le couple (D, G), qui doit être fait en trois unités de temps et les agents se rencontrent au noeud r. Ce qui donne la même configuration qu'à la phase précédente. Nous pouvons

continuer ainsi jusqu'à l'avant dernière phase. Si les agents n'explorent pas le noeud D, on se retrouvera avec la configuration montrée à la figure 5.9. À ce moment-ci, les agents peuvent explorer le couple (D, F) en trois unités de temps et poursuivre avec l'exploration du noeud H en trois unités de temps, ce qui donne un temps de $3k - 1$. Une autre solution est d'explorer le noeud D en deux unités de temps et le couple (F, H) en quatre unités de temps. Ces deux solutions donnent un temps de $3k - 1$ unités de temps.

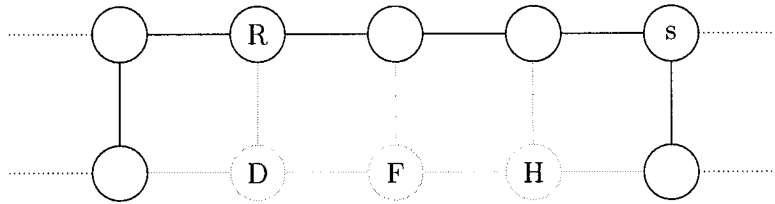


FIG. 5.9 – Tore 2 par k : Troisième configuration, avant dernière phase

Si les agents explorent le couple (D, E), ce qui doit se faire en trois unités de temps avec comme point de rencontre le noeud r'', le noeud C devra être exploré à la phase suivante, sans quoi il sera isolé et ne pourra plus être exploré avec les types de phases admises. Cependant, si les agents explorent le noeud C avec un autre noeud durant la phase suivante, ils devront le faire en quatre unités de temps et ne pourront se positionner pour une phase légère.

Finalement, si les agents explorent le noeud C en deux unités de temps, ils se retrouvent dans une configuration identique à la figure 5.4. On peut utiliser l'induction décrite dans le lemme 5.9 pour les phases suivantes jusqu'à la dernière. Pour la dernière phase, on se retrouve avec exactement la même configuration qu'à la figure 5.5, mais nous avons montré dans le lemme 5.9 qu'il était impossible de faire cette exploration en moins de quatre unités de temps. Donc nous obtenons un temps d'au moins $3k - 1$.

Pour la **quatrième et la cinquième configuration**, qui sont montrées à la figure 5.6(d) et 5.6(e), nous pouvons observer que si l'on enlève le point de départ de l'algorithme s, on peut obtenir la cinquième configuration à partir de la quatrième confi-

guration à l'aide de symétrie. Par conséquent, nous n'allons analyser que la quatrième configuration et les résultats pourront être utilisé pour la cinquième configuration.

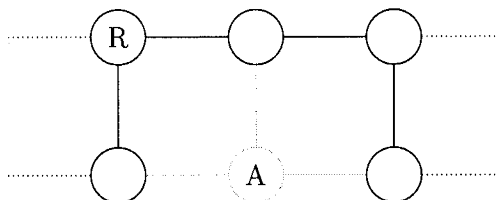


FIG. 5.10 – Tore 2 par k : Dernière phase pour les configurations 4,5 et 6

Les noeuds A, B, C et D peuvent être explorés en troisième phase. Les couples (A, C) et (A, D) peuvent être explorés en quatre unités de temps et la rencontre se fait au noeud R, ce qui rend impossible une phase légère. De même pour les couples (A, B) et (B, C), où la rencontre est au noeud r. Le couple (B, D) ne peut pas être exploré en moins de cinq unités de temps. Il ne reste que le couple (C, D), qui doit être exploré en trois unités de temps avec le noeud r' comme point de rencontre. Encore une fois, on se retrouve dans une configuration identique à la figure 5.4. On peut utiliser l'induction décrite dans le lemme 5.9 pour les phases suivantes jusqu'à la dernière. La dernière phase aura la configuration montrée à la figure 5.10, avec le noeud A restant à explorer. Les agents devront utiliser trois unités de temps pour explorer le dernier noeud, ce qui donne un temps d'au moins $3k - 1$ pour l'algorithme.

Pour la **sixième configuration**, qui est montré à la figure 5.6(f), les noeuds A, B, C et D peuvent être explorés en troisième phase. Si les agents explorent le couple (A, B) ou (A, C), ils doivent utiliser quatre unités de temps et se rencontrer au noeud r, où ils ne peuvent plus continuer avec une phase légère. De même pour le couple (C, D), où le point de rencontre est au noeud r''. Les couples (A, D) et (B, D) ne peuvent pas être explorés en moins de cinq unités de temps.

Si les agents explorent le noeud C en deux unités de temps, ils se retrouvent dans une situation où seuls les couples (A, B), (B, F) et (D, F) peut être explorés en moins de cinq unités de temps. Ils peuvent être explorés en quatre unités de temps, mais dans tout les cas le point de rencontre ne permet pas une phase légère.

Finalement, les agents peuvent explorer le couple (B, C) en trois unités de temps et se rencontrer au noeud r'. Ils se retrouvent par symétrie dans une configuration identique à la figure 5.4. On peut utiliser l'induction décrite dans le lemme 5.9 pour les phases suivantes jusqu'à la dernière. La dernière phase aura la configuration montrée à la figure 5.10, avec le noeud A restant à explorer. Les agents devront utiliser trois unités de temps pour explorer le dernier noeud, ce qui donne un temps d'au moins $3k-1$ pour l'algorithme. □

THÉORÈME 5.2. *L'algorithme $\mathcal{A}$ qui explore le tore 2 par k en temps $3k-2$ n'existe pas.*

Démonstration : Selon les lemmes 5.9 et 5.10, le temps minimal de $3k-2$ ne peut pas être atteint lorsque l'on explore trois ou quatre noeuds dans les deux premières phases. Pourtant selon la démonstration du théorème 5.1, si un algorithme explore le tore en temps $3k-2$, un de ces deux cas doit arriver. □

COROLLAIRE 5.1. *Le temps requis pour un algorithme RTN dans un tore de grandeur 2 par k est au moins de $3k-1$.*

ALGORITHME 5.1. *Tore2k*

On suppose sans perte de généralité que $s = x_0$.

fourche(y_0,x_1) ;
pour $i := 1$ **à** $k-2$ **faire**
  marche(x_i) ;
  fourche(y_i,x_{i+1}) ;
marche(x_{k-1}) ;
sonder(y_{k-1}) ;

THÉORÈME 5.3. *L'algorithme Tore2k est optimal.*

Démonstration : La procédure fourche du départ prend deux unités de temps. Ensuite, chaque itération de la boucle prend trois unités de temps, une pour la marche vers le noeud adjacent et deux pour la fourche. La boucle fait $k-2$ itérations. Ensuite, trois unités de temps sont utilisées, une pour la marche vers le noeud adjacent et deux autres pour sonder le dernier noeud à visiter. Ce qui donne un total de $2+3(k-2)+3 = 3k-1$ unités de temps. Selon le corollaire 5.1 notre algorithme est optimal. □

Chapitre 6

Résultats pour le tore de grandeur m par k

On considère un tore de grandeur m par k, où $m \geq 3$ et $m \leq k$, défini comme suit : $T = \{V, E\}$ où $V = \{(i,0), ..., (i, k-1) : i = 0, ..., m-1\}$, $E = \{\{(i,j), ((i+1), j)\} : i = 0, ..., m-1 \wedge j = 0, ..., k-1\}\} \cup \{\{(i,j), (i, (j+1))\} : i = 0, ..., m-1 \wedge j = 0, ..., k-1\}$, et un noeud s qui est le point de départ des deux agents. Les opérations arithmétiques sur l'indice i sont faites modulo m et les opérations arithmétiques sur l'indice j sont faites modulo k.

Nous prouverons des bornes inférieures sur le temps de fonctionnement de n'importe quel algorithme RTN dans les tores ci-dessus et nous présenterons des algorithmes dont la performance est proche de ces bornes. Plus précisément pour les tores 3 par k notre algorithme sera optimal et dans le cas général nous présenterons un algorithme d'approximation avec taux 1.3 qui est asymptotiquement optimal lorsque la plus petite dimension du tore n'est pas bornée.

LEMME 6.1. *Une phase légère $p \geq 2$ ne peut pas être suivie d'une autre phase légère.*

Démonstration : Les noeuds d'un tore de grandeur m par k, où $m \geq 3$, ont un degré quatre. Lorsque le territoire connu est ≥ 2, les noeuds ont au maximum trois noeuds adjacents inconnus. Pour qu'une phase légère soit possible, les agents doivent débuter la phase sur un noeud ayant au moins deux noeuds adjacents inconnus. Puisque nous n'avons qu'un maximum de trois noeuds inconnus au début d'une phase $p \geq 2$, deux phases légères consécutives sont impossibles. □

Un noeud i est ouvert à la phase $p \geq 2$, lorsque ce noeud i fait parti du territoire connu et qu'il n'est adjacent qu'à un seul noeud du territoire connu. Un segment d'exploration $E(p)$ est défini comme toutes les phases d'un algorithme RTN jusqu'à la phase p.

LEMME 6.2. *Soit un segment d'exploration $E(p)$, où $p \geq 3$. Il existe un segment d'exploration $E'(p)$ où la phase p ne débute pas sur un noeud ouvert et qui donne le même territoire connu, le même point de rencontre et le même temps que $E(p)$.*

Démonstration : Soit i le point de rencontre à la phase $p-1$. Il existe un noeud j adjacent au noeud i et qui est dans le territoire connu à la phase $p-1$. Si le noeud i est ouvert, le dernier mouvement des agents à la phase $p-1$ est une marche du noeud j à i. Supposons que dans le segment d'exploration $E'(p)$ le point de rencontre à la phase $p-1$ est au noeud j et que les agents marchent de j vers i au début de la phase p. Toutes les autres actions effectuées à la phase p restent identiques dans $E'(p)$. Si l'on compare $E(p)$ et $E'(p)$, la phase $p-1$ prend une unité de temps en moins et la phase p en prend une de plus. Le point de rencontre à la phase p, le territoire connu et le nombre d'unités de temps utilisé sont identiques dans les deux segments d'explorations. □

Par la suite nous supposons, sans perte de généralité, que les phases $p \geq 3$ d'un algorithme RTN ne débutent pas sur un noeud ouvert.

LEMME 6.3. *Une phase légère $p \geq 3$ ne peut pas être suivie d'une phase n'explorant qu'un noeud et utilisant deux unités de temps.*

Démonstration : Soit une phase légère $p \geq 3$ où le point de rencontre est le noeud i. Par notre supposition, le noeud i n'est pas ouvert à la phase p. Suite à la phase p, le noeud i n'a plus de noeud adjacent inconnu, donc les agents ne peuvent pas explorer un noeud en deux unités de temps durant la phase $p+1$. □

Par le même raisonnement, on peut prononcer le lemme suivant :

LEMME 6.4. *Une phase $p \geq 3$ explorant un noeud en deux unités de temps ne peut pas être suivie d'une phase légère.*

LEMME 6.5. *La séquence des phases : [légère, normale, légère] est impossible à partir de la phase*

i) $p \geq 2$ si la première phase de l'algorithme est légère.

ii) $p \geq 3$ *autrement.*

Démonstration : Si la première phase de l'algorithme est une phase légère, les deux agents débutent la deuxième phase sur un noeud qui n'est pas ouvert, autrement nous sommes à une phase $p \geq 3$ et les agents commencent sur un noeud n'étant pas ouvert. Supposons R le point de départ de la phase p, présenté à la figure 6.1. Pour permettre une phase légère à la phase p, R doit avoir deux noeuds adjacents inconnus : A et B Le point de rencontre de cette première phase est le noeud R. Rappelons qu'une phase normale est une phase explorant deux noeuds en trois unités de temps. Pour que la phase $p+1$ soit normale, les agents doivent se déplacer vers un même noeud ayant deux noeuds adjacents inconnus, C et D, et explorer ces noeuds. Supposons sans perte de généralité que ce noeud est le noeud A. La phase $p+2$ débute sur un noeud n'ayant qu'un seul noeud adjacent inconnu E. Elle ne peut donc pas être légère.

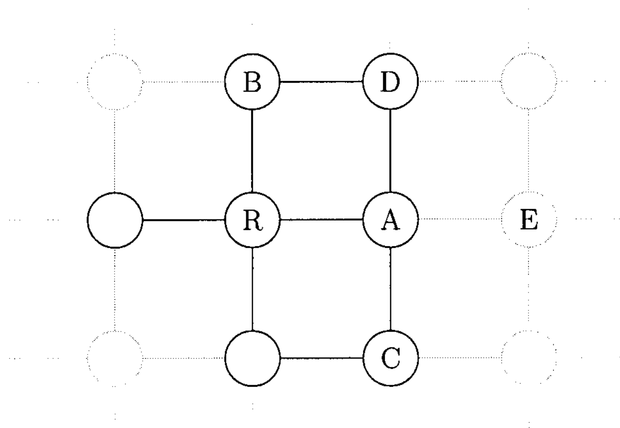


FIG. 6.1 – Tore m par k : Séquence des phases : légère, normale, légère

□

LEMME 6.6. *Soit x le nombre de phases $p \geq 2$ si la première phase de l'algorithme est légère et soit x le nombre de phases $p \geq 3$ autrement. Alors le temps d'exploration de ces x phases est au moins de :*

i) $\frac{4}{3}x$ *unités de temps, si x modulo* $6 = 0$

ii) $\frac{4}{3}x + \frac{2}{3}$ *unités de temps, si x modulo* $6 = 1$

iii) $\frac{4}{3}x - \frac{2}{3}$ *unités de temps, si* x *modulo* $6 = 2$
iv) $\frac{4}{3}x + 1$ *unités de temps, si* x *modulo* $6 = 3$
v) $\frac{4}{3}x - \frac{1}{3}$ *unités de temps, si* x *modulo* $6 = 4$
vi) $\frac{4}{3}x + \frac{1}{3}$ *unités de temps, si* x *modulo* $6 = 5$

Démonstration : Considérons la suite des x phases, en utilisant le symbole l pour une phase légère, le symbole L pour une phase lourde et le symbole a pour une phase autre que légère. Subdivisons ces phases en séquences comprenant trois phases chacune. Rappelons qu'en vue du lemme 6.1, une phase légère ne peut pas être suivie par une autre phase légère. Aussi, selon le lemme 6.5, la séquence des phases lnl est impossible. Nous ne pouvons donc avoir que les séquences de phases suivantes : aaa, laa, lLl, ala et aal.

Par définition, une phase légère explore deux noeuds en deux unités de temps et une phase lourde explore deux noeuds en au moins quatre unités de temps. Pour les phases autre que légère, elles peuvent explorer soit un noeud, soit deux noeuds. Si cette phase explore deux noeuds, elle prend au moins trois unités de temps et le temps moyen d'exploration d'un noeud est d'au moins $\frac{3}{2}$ unités de temps. Si elle explore un seul noeud, elle prends au moins deux unités de temps et le temps moyen d'exploration d'un noeud est d'au moins 2 unités de temps, ce qui est pire que lorsqu'elle explore deux noeuds.

Le temps moyen d'exploration d'un noeud pour la séquence aaa est d'au moins $\frac{3}{2}$ et d'au moins $\frac{4}{3}$ pour tous les autres. Donc le temps moyen d'exploration d'un noeud pour toutes ces séquences est d'au moins $\frac{4}{3}$ unités de temps.

Cependant, nous avons vu que lorsqu'une phase a n'explore qu'un noeud, le temps moyen d'exploration d'un noeud ne peut pas être de $\frac{4}{3}$. Donc tous ces segments doivent explorer six noeuds pour avoir un temps moyen d'exploration d'un noeud d'au moins $\frac{4}{3}$. Le dernier segment S peut alors avoir de un à six noeuds restant à explorer, tout dépendant du reste de la division de x par six.

Utilisons le symbole u pour une phase explorant un noeud, le symbole d pour une phase explorant deux noeuds, le symbole a_u pour une phase autre explorant un noeud en plus de deux unités de temps et le symbole a_d pour une phase autre explorant deux noeuds. Gardons en vue les lemmes 6.3 et 6.4 qui indiquent qu'une phase légère ne peut pas être précédé ou suivie d'une phase explorant un noeud en deux unités de temps.

S'il y a un ou deux noeuds dans le segment S, deux unités de temps au moins sont nécessaires pour l'exploration.

S'il y a trois noeuds dans le segment S, nous pouvons effectuer les séquences : $a_d u$, ua_d, la_u, $a_u l$ ou uuu. La séquence uuu prend au moins six unités de temps et les autres prennent au moins cinq unités de temps.

S'il y a quatre noeuds dans le segment S, nous pouvons effectuer les séquences : $a_d a_d$, la_d, $a_d l$, $la_u u$, $a_d uu$, $a_u la_u$, $ua_d u$, $ua_u l$ ou uua_d. Les séquences la_d et $a_d l$ prennent au moins cinq unités, la séquence $a_d a_d$ prends au moins six unités de temps, les séquences $la_u u$, $a_d uu$, $ua_d u$, $ua_u l$ et uua_d prennent au moins sept unités de temps et la séquence $a_u la_u$ prends au moins huit unités de temps.

Finalement s'il y a cinq noeuds dans le segment S, nous pouvons effectuer les séquences : $da_d u$, $a_d la_u$, $da_u d$, $a_d ua_d$, $a_u la_d$ ou $ua_d d$. Les séquences $da_d u$, $da_u d$ et $ua_d d$ prennent au moins sept unités de temps et les séquences $a_d la_u$, $a_d ua_d$ et $a_u la_d$ prennent au moins huit unités de temps.

Le temps d'exploration pour toutes les phases $p \geq 2$ si la première phase de l'algorithme est légère, $p \geq 3$ autrement, est au moins de :

i) $\frac{4}{3}x$ unités de temps, si x modulo $6 = 0$
ii) $\frac{4}{3}(x-1)+2 = \frac{4}{3}x + \frac{2}{3}$ unités de temps, si x modulo $6 = 1$
iii) $\frac{4}{3}(x-2)+2 = \frac{4}{3}x - \frac{2}{3}$ unités de temps, si x modulo $6 = 2$
iv) $\frac{4}{3}(x-3)+5 = \frac{4}{3}x + 1$ unités de temps, si x modulo $6 = 3$
v) $\frac{4}{3}(x-4)+5 = \frac{4}{3}x - \frac{1}{3}$ unités de temps, si x modulo $6 = 4$
vi) $\frac{4}{3}(x-5)+7 = \frac{4}{3}x + \frac{1}{3}$ unités de temps, si x modulo $6 = 5$

□

THÉORÈME 6.1. *Le temps requis pour un algorithme RTN dans un tore de grandeur m par k, où $m \geq 3$, $m \leq k$ et $n = m \times k$, est au moins de :*

i) $\frac{4}{3}n - 1$ unités de temps, si n modulo $6 = 0$
ii) $\frac{4}{3}n - \frac{7}{3}$ unités de temps, si n modulo $6 = 1$
iii) $\frac{4}{3}n - \frac{5}{3}$ unités de temps, si n modulo $6 = 2$
iv) $\frac{4}{3}n - 2$ unités de temps, si n modulo $6 = 3$
v) $\frac{4}{3}n - \frac{4}{3}$ unités de temps, si n modulo $6 = 4$
vi) $\frac{4}{3}n - \frac{8}{3}$ unités de temps, si n modulo $6 = 5$

Démonstration : Le lemme 6.6 nous montre le temps minimal d'exploration pour toutes les phases $p \geq 2$ si la première phase de l'algorithme est légère, $p \geq 3$ autrement. Dans le cas où nous avons $p \geq 3$ phases, les deux premières phases peuvent explorer entre deux

et quatre noeuds. Si nous n'explorons que deux noeuds, le temps moyen d'exploration d'un noeud sera supérieur à celui où la première phase est légère. Donc, nous ne prenons pas ce cas en considération. Le temps minimal pour explorer trois noeuds dans les deux premières phases est de quatre unités de temps. Le temps minimal pour explorer quatre noeuds dans les deux premières phases est de cinq unités de temps si on n'utilise pas de phase légère comme première phase. Le tableau 6.1 montre le temps minimal requis pour les différentes situations, le meilleur temps étant en caractère gras.

n modulo 6	Première phase légère	Trois noeuds dans les deux premières phases	Quatre noeuds dans les deux premières phases
0	$\mathbf{\frac{4}{3}n - 1}$	$\frac{4}{3}n - \frac{2}{3}$	$\frac{4}{3}n + \frac{2}{3}$
1	$\mathbf{\frac{4}{3}n - \frac{7}{3}}$	$\frac{4}{3}n + 1$	$\frac{4}{3}n - \frac{2}{3}$
2	$\mathbf{\frac{4}{3}n - \frac{5}{3}}$	$\frac{4}{3}n - \frac{1}{3}$	$\frac{4}{3}n + 1$
3	$\mathbf{\frac{4}{3}n - 2}$	$\frac{4}{3}n + \frac{1}{3}$	$\frac{4}{3}n - \frac{1}{3}$
4	$\mathbf{\frac{4}{3}n - \frac{4}{3}}$	$\frac{4}{3}n$	$\frac{4}{3}n + \frac{1}{3}$
5	$\mathbf{\frac{4}{3}n - \frac{8}{3}}$	$\frac{4}{3}n + \frac{2}{3}$	$\frac{4}{3}n$

TAB. 6.1 – Borne inférieure sur tore de grandeur m par k

□

ALGORITHME 6.1. *Tore3k*

On suppose sans perte de généralité que $s = (1, 0)$.

fourche($(1, k-1)$,$(0, 0)$) ;
pour $i := 0$ **à** $\lfloor n/6 \rfloor - 2$ **faire**
  fourche($(2, 2i)$,$(1, 2i+1)$) ;
  marche($(1, 2i+1)$) ;
  fourche($(2, 2i+1)$,$(1, 2i+2)$) ;
  fourche($(0, 2i+1)$,$(0, 2i+2)$) ;
si n modulo 6 = 3 **alors**
  fourche($(2, k-3)$,$(1, k-2)$) ;
  marche($(1, k-2)$) ;
  fourche($(2, k-2)$,$(1, k-2)$) ;
  marche($(1, k-1)$) ;
  fourche($(0, k-1)$,$(2, k-1)$) ;

autrement
fourche($(2, k-2)$,$(0, k-1)$) ;
sonder($(2, k-1)$) ;

THÉORÈME 6.2. *L'algorithme Tore3k est optimal.*

Démonstration : La procédure fourche du départ prend deux unités de temps. Ensuite, chaque itération de la boucle prend huit unités de temps et explore six noeuds, ce qui donne un temps moyen de $\frac{4}{3}$ unités de temps par noeud.

Dans le cas où n modulo $6 = 3$ on effectue après la boucle trois fourches et une marche ce qui prend huit unités de temps et explore six noeuds. Ce qui donne un temps moyen de $\frac{4}{3}$ unités de temps par noeud. Nous explorons tous les noeuds sauf trois (les deux premiers et le point de départ), avec un temps moyen de $\frac{4}{3}$ unités de temps par noeud, ce qui donne un total de $\frac{4}{3}(n-3)+2 = \frac{4}{3}n-2$ unités de temps. Dans les autres cas, c'est-à-dire lorsque n modulo $6 = 0$, on effectue une fourche qui prend trois unités de temps et sonder le dernier noeud prend deux unités de temps, ce qui donne un total de $\frac{4}{3}(n-6)+2+5 = \frac{4}{3}n-1$ unités de temps. Selon le théorème 6.1 notre algorithme est optimal. □

Nous présentons maintenant un algorithme qui résout le problème RTN pour tous les tores. Lorsque $m \geq 2$, on considère m modulo 3 et selon le résultat, différentes avenues sont utilisés. Lorsque le résultat du modulo est égal à 0, on divise le tore en sous-tores de grandeur 3 par k, et on exécute l'algorithme Tore3k sur chacun d'eux. Entre chaque sous-tore, les agents explorent les noeuds pour se placer en position pour relancer l'algorithme Tore3k. Si le résultat du modulo est différent de 0, le tore est encore divisé en sous-tores, mais selon le résultat du modulo, le ou les derniers sous-tores seront de grandeur 2 par k. Si le résultat du modulo est de 1, les deux derniers sous-tores seront des tores de grandeur 2 par k. Si le résultat du modulo est de 2, seulement le dernier tore sera de grandeur 2 par k. Sur ces tores, on lance l'algorithme Tore2k. Si $m = 1$, on utilise l'algorithme Anneau.

ALGORITHME 6.2. *Tore*

On suppose sans perte de généralité que $s = (1, 0)$.

si $m = 1$ **alors**
Anneau

autrement
    fourche((0, 0),(1, $k-1$)) ;
  cas où : m modulo 3 = 0
      exploreTore(1,0,$m/3$,0)
  cas où : m modulo 3 = 1
      exploreTore(1,0,$\lfloor m/3 \rfloor - 1$,2)
  cas où : m modulo 3 = 2
      exploreTore(1,0,$\lfloor m/3 \rfloor$,1)

Dans la procédure suivante (i, j) est le noeud où les agents commencent la procédure, o est le nombre de sous-tores de grandeur 3 par k restant à explorer et p sert à déterminer le nombre de sous-tores de grandeur 2 par k à explorer.

PROCEDURE 6.1. *exploreTore(i,j,o,p)*

Les opérations arithmétiques sur l'indice j sont faites $+k$ modulo k

si $o \geq 1$
    pour $q := 0$ **à** $\lfloor k/2 \rfloor - 2$ **faire**
      fourche(($i+1, j+2q$),($i, j+2q+1$)) ;
      marche(($i, j+2q+1$)) ;
      fourche(($i+1, j+2q+1$),($i, j+2q+2$)) ;
      fourche(($i-1, j+2q+1$),($i-1, j+2q+2$)) ;
    si k modulo 2 = 1 **alors**
      fourche(($i+1, j-3$),($i, j-2$)) ;
      marche(($i, j-2$)) ;
      fourche(($i+1, j-2$),($i, j-2$)) ;
      marche(($i, j-1$)) ;
      fourche(($i-1, j-1$),($i+1, j-1$)) ;
      si $o > 1$ **ou bien** $p \neq 0$ **alors**
        marche(($i+1, j-1$)) ;
        sonder(($i+2, j-1$)) ;
    autrement
      fourche(($i+1, j-2$),($i+1, j-1$)) ;
      si $o > 1$ **ou bien** $p \neq 0$ **alors**
        fourche(($i-1, j-1$),($i+2, j-1$)) ;
      autrement

```
        sonder((i − 1, j − 1)) ;
      si o > 1 ou bien p ≠ 0 alors
        marche((i + 2, j − 1)) ;
        sonder((i + 3, j − 1)) ;
        marche((i + 3, j − 1)) ;
        sonder((i + 3, j − 2)) ;
        exploreTore(i + 3,j − 1,o − 1,p) ;
  autrement
    cas où : p = 2
        pour q := 1 à k − 2 faire
          marche((i, j − q)) ;
          fourche((i − 1, j − q),(i, j − q − 1)) ;
        marche((i, j + 1)) ;
        fourche((i − 1, j + 1),(i + 1, j + 1)) ;
        marche((i + 1, j + 1) ;
        sonder((i + 2, j + 1) ;
        marche((i + 2, j + 1) ;
        sonder((i + 2, j)) ;
        exploreTore(i + 2,j + 1,o,1) ;
    cas où : p = 1
        pour q := 1 à k − 2 faire
          marche((i, j − q)) ;
          fourche((i − 1, j − q),(i, j − q − 1)) ;
        marche((i, j + 1)) ;
        sonder((i − 1, j + 1)) ;
```

THÉORÈME 6.3. *L'algorithme Tore est asymptotiquement optimal lorsque sa plus petite dimension n'est pas bornée et dans le cas général il a un taux d'approximation 1.3*

Démonstration : Excluons le cas où $m = 1$, où nous savons que notre algorithme est optimal. Pour les autres cas, nous avons six situations différentes : lorsque m modulo 3 donne un résultat de 0, 1 ou 2 et chacun de ces cas se divise en deux : lorsque k est pair ou impair.

Dans tous les cas, une fourche de deux unités de temps est utilisé avant de commencer la procédure récursive 6.1. Selon si k est pair ou impair, la procédure principale qui est

effectuée lorsque $o \geq 1$ prends un temps différent. Cette procédure explore un sous-tore de grandeur 3 par k et les agents se place prêt pour l'exploration de prochain sous-tore, sauf lorsqu'il n'y a plus de sous-tore à explorer. Lorsque k est pair, cette procédure prend $4k + 4$ unités de temps et $4k - 3$ unités de temps s'il n'y a plus de sous-tore à explorer. Si k est impair, elle prend $4k + 5$ unités de temps et $4k - 4$ unités de temps s'il n'y a plus de sous-tore à explorer.

Lorsque m modulo 3 donne un résultat de 2, la procédure principale, quand $o \geq 1$, est exécutée $(m-2)/3$ fois et le cas où $p = 1$ est exécuté une fois. Ce cas prend $3k - 3$ unités de temps. Lorsque m modulo 3 donne un résultat de 1, la procédure principale, quand $o \geq 1$, est exécutée $(m-1)/3 - 1$ fois, le cas où $p = 2$ est exécuté une fois et le cas où $p = 1$ est exécuté aussi une fois. Le cas où $p = 2$ prend $3k + 3$ unités de temps. Finalement, lorsque m modulo 3 donne un résultat de 0, la procédure principale, quand $o \geq 1$, est exécutée $m/3 - 1$ fois et une dernière fois avec le paramètre $p = 0$.

Le tableau 6.2 montre les différents temps possibles de l'algorithme lorsque $m \geq 2$.

m mod 3	k mod 2	selon m et k
0	0	$\frac{4}{3}mk + \frac{4}{3}m - 5$
	1	$\frac{4}{3}mk + \frac{5}{3}m - 7$
1	0	$\frac{4}{3}mk + \frac{2}{3}k + \frac{4}{3}m - \frac{10}{3}$
	1	$\frac{4}{3}mk + \frac{2}{3}k + \frac{5}{3}m - \frac{14}{3}$
2	0	$\frac{4}{3}mk + \frac{1}{3}k + \frac{4}{3}m - \frac{11}{3}$
	1	$\frac{4}{3}mk + \frac{1}{3}k + \frac{5}{3}m - \frac{13}{3}$

TAB. 6.2 – Temps de l'algorithme Tore

Lorsque $m = 1, 2, 3$, l'algorithme Tore est optimal, puisqu'il est équivalent aux algorithmes Anneau, Tore2k et Tore3k respectivement.

Lorsque $m \geq 4$, l'algorithme Tore n'est pas optimal. Tout de même, nous allons montrer que sa performance est alors proche de l'optimale.

Nous savons que $m \leq k$ et que $n = mk$, alors $\frac{4}{3}mk + \frac{5}{3}m \leq \frac{4}{3}n + \frac{5}{3}\sqrt{n}$. Lorsque le résultat de m modulo 3 égal à 0 et que $m \geq 4$, la borne inférieure est au moins $\frac{4}{3}n - 2$. Puisque $\lim_{n\to\infty} \frac{\frac{4}{3}n+\frac{5}{3}\sqrt{n}}{\frac{4}{3}n-2} = 1$, l'algorithme est asymptotiquement optimal.

Nous savons que $m \leq k$ et que $k = \frac{n}{m}$. Le temps de fonctionnement de l'algorithme Tore est toujours au plus $\frac{4}{3}n + \frac{2}{3}k + \frac{5}{3}m \leq \frac{4}{3}n + 3k = \frac{4}{3}n + 3\frac{n}{m}$. Considérons une famille

de tores $m(n)$ par $k(n)$ tels que $m(n) \cdot k(n) = n$, $m(n) \leq k(n)$, avec $m(n)$ non-borné, c'est-à-dire $\lim_{n \to \infty} m(n) = \infty$. Puisque dans ce cas $\lim_{n \to \infty} \frac{\frac{4}{3}n + 3\frac{n}{m(n)}}{\frac{4}{3}n - 3} = 1$, l'algorithme Tore est asymptotiquement optimal si $m(n)$ est non-borné.

Dans le cas général nous allons montrer que l'algorithme Tore a un taux d'approximation de 1.3.

La proportion du temps de fonctionnement de l'algorithme Tore par rapport à la borne inférieure est une fonction décroissante des paramètres k et m. Puisque nous savons que notre algorithme est optimal pour $m = 1, 2, 3$, dans chaque cas nous devons utiliser le plus petit $m \geq 4$ pour évaluer le taux d'approximation de notre algorithme. Aussi, nous savons que $m \leq k$. Le tableau 6.3 montre le taux d'approximation dans chaque cas. Le maximum est donc 1.3, ce qui devient le taux d'approximation de l'algorithme Tore.

m mod 3	k mod 2	m	k	différence avec la borne inférieur en pire cas
0	0	6	6	$\frac{\frac{4}{3}mk+\frac{4}{3}m-5}{\frac{4}{3}n-1} = \frac{51}{47}$
	1	6	7	$\frac{\frac{4}{3}mk+\frac{5}{3}m-7}{\frac{4}{3}n-1} = \frac{59}{55}$
1	0	4	4	$\frac{\frac{4}{3}mk+\frac{2}{3}k+\frac{4}{3}m-\frac{10}{3}}{\frac{4}{3}n-\frac{4}{3}} = \frac{13}{10}$
	1	4	5	$\frac{\frac{4}{3}mk+\frac{2}{3}k+\frac{5}{3}m-\frac{14}{3}}{\frac{4}{3}n-\frac{5}{3}} = \frac{32}{25}$
2	0	5	5	$\frac{\frac{4}{3}mk+\frac{1}{3}k+\frac{5}{3}m-\frac{13}{3}}{\frac{4}{3}n-\frac{7}{3}} = \frac{39}{31}$
	1	5	6	$\frac{\frac{4}{3}mk+\frac{1}{3}k+\frac{4}{3}m-\frac{11}{3}}{\frac{4}{3}n-1} = \frac{45}{39}$

TAB. 6.3 – Algorithme Tore : Différence avec la borne inférieur en pire cas

□

Chapitre 7

Conclusion

Nous avons présenté des algorithmes qui résolvent le problème de recherche de trou noir dans les réseaux informatiques avec la topologie de tore. Pour les anneaux, les tores de grandeur 2 par k et les tores de grandeur 3 par k, nous avons trouvé des algorithmes optimaux. Dans le cas général nous présentons un algorithme d'approximation avec taux 1.3 qui est asymptotiquement optimal lorsque la plus petite dimension du tore n'est pas bornée. Le temps de fonctionnement de tous ces algorithmes est linéaire selon le nombre de noeuds.

Il serait intéressant de savoir quel est l'écart exact entre la performance de notre algorithme et la performance optimale. Le problème de trouver l'algorithme RTN optimal pour un tore arbitraire reste aussi ouvert. Il serait également intéressant de trouver des algorithmes pour d'autres topologies couramment utilisées dans les réseaux informatiques. Des variantes du problème RTN avec un nombre plus grand d'agents et de trous noirs seraient aussi importants à étudier.

Annexe A

Code de anneau.java

```
/**
* Ce programme utilise la programmation dynamique pour trouver
* le temps minimal l'exploration des anneaux jusqu'à 8 noeuds à
* l'aide d'agents mobiles. Il génère des tableaux en format LaTeX.
*
* @author Eric Vachon
*/

import java.util.*;

class anneau{


  public static int size = 9;
  public static long tab[][] = new long[size][(int)Math.ceil((size-1)/2.0)];
  public static String tabWay[][] = new String[size][(int)Math.ceil((size-1)/2.0)];

  public static boolean added[][] = new boolean[size][(int)Math.ceil((size-1)/2.0)];

  public static void main(String args[]){

    for(int i = 1; i < size; i++){
      for(int j = 0; j < Math.ceil((size-1)/2.0); j++){
        tab[i][j] = Integer.MAX_VALUE;
        added[i][j] = false;
      }
    }
    tab[1][0] = 0;
    tabWay[1][0] = new String("(-,-)");
    added[0][0] = true;
    showtab();
    added[0][0] = false;;
    for(int i = 1; i < size - 1; i++){
      for(int j = 0; j < (int)Math.ceil((size-1)/2.0); j++){
```

```
if(Math.ceil(i/2.0) - 1 >= j){
  System.out.println("\\begin{itemize}");
  System.out.println("\\item {\\bf Tests pour l'état (" + i + "," + j + ")}");
  System.out.println("\\begin{itemize}");
  if(i != 1){
    for(int k = 0; k < (int)Math.ceil((size-1)/2.0); k++){
      if(k < Math.ceil((i+1)/2.0)){
        System.out.print("\\item Ajout d'un noeud supplémentaire : ");
        System.out.print("(" + (i+1) + "," + k + ") = ");
        if(tab[i+1][k] == Integer.MAX_VALUE){
          System.out.print("$\\infty$");
        }else{
          System.out.print(tab[i+1][k]);
        }
        long temp1 = tab[i][j] + j + 2 + Math.abs(k-1);
        if(tab[i+1][k] > temp1){
          System.out.println(" > " + temp1 + " {\\bf change}");
          tabWay[i+1][k] = new String("("+i+","+j+")" );
          tab[i+1][k] = temp1;
          added[i+1][k] = true;
        }else{
          System.out.println(" $\\leq$ " + temp1 + " {\\bf garde}");
        }
      }
    }
  }else{
      System.out.print("\\item Ajout d'un noeud supplémentaire : ");
      System.out.print("(" + (i+1) + "," + 0 + ") = $\\infty$");
      System.out.println(" > " + 2 + " {\\bf change}");
      tabWay[2][0] = new String("(1,0)");
      tab[2][0] = 2;
      added[2][0] = true;
  }
  if(i < size - 2){
  for(int k = 0; k < (int)Math.ceil((size-1)/2.0); k++){
      if(k < Math.ceil((i+2)/2.0)){
        System.out.print("\\item Ajout de deux noeuds supplémentaires : ");
        System.out.print("(" + (i+2) + "," + k + ") = ");
        if(tab[i+2][k] == Integer.MAX_VALUE){
          System.out.print("$\\infty$");
        }else{
          System.out.print(tab[i+2][k]);
        }
        long temp2 = tab[i][j] + i + 1 + Math.abs(k-(j+1));
        if(tab[i+2][k] > temp2){
          System.out.println(" > " + temp2 + " {\\bf change}");
          tabWay[i+2][k] = new String("("+i+","+j+")");
          tab[i+2][k] = temp2;
          added[i+2][k] = true;
        }else{
          System.out.println(" $\\leq$ " + temp2 + " {\\bf garde}");
```

```
            }
          }
        }
      }
      System.out.println("\\end{itemize}");
      showtab();
      System.out.println("\\end{itemize}");
    }else{
      tab[i][j] = -1;
    }
  }
}
showtab();
}

public static void showtab(){
  boolean ok = false;
  for(int i = 1; i < size; i++){
    for(int j = 0; j < Math.ceil((size-1)/2.0); j++){
      if(added[i][j]) ok = true;
    }
  }
  if(ok){
    //System.out.println("\\begin{table}[h]");
    //System.out.println("\\centering");
    System.out.println("\\begin{center}");
    System.out.print("  \\begin{tabular}{|l|");
    for(int i = 0; i < Math.ceil((size-1)/2.0); i++){
        System.out.print("|l");
    }
    System.out.println("|}");
    System.out.println("    \\hline");
    System.out.print("    ");
    for(int i = 0; i < Math.ceil((size-1)/2.0); i++){
        System.out.print("& " + i + " ");
    }
    System.out.println("\\\\");
    System.out.println("    \\hline \\hline");
    for(int i = 1; i < size; i++){
      System.out.print("    ");
      System.out.print(i + " &");
      for(int j = 0; j < Math.ceil((size-1)/2.0); j++){
        if(j >= ((i+1)/2)){
          System.out.print(" - ");
        }else if(tab[i][j] == -1 || tab[i][j] == Integer.MAX_VALUE){
          System.out.print(" $\\infty$ ");
        }else{
          if(added[i][j]){
            added[i][j] = false;
            System.out.print(" \\boldmath$" + tab[i][j] + "$ " );
          }else{
```

```
            System.out.print(" $" + tab[i][j] + "$ " );
          }
        }
        if(j != Math.ceil((size-1)/2.0) - 1 ){
          System.out.print("&");
        }
      }
      System.out.println("\\\\");
      System.out.println("    \\hline");

    }
    System.out.println("  \\end{tabular}");
    System.out.println("\\end{center}");
    //System.out.println("\\end{table}");
  }
 }

}
```

Bibliographie

[1] AWERBUCH, B., AND BETKE, M. Piecemeal graph exploration by a mobile robot. *Information and Computation 152*, 2 (1999), 155–172.

[2] BAR-ELI, E., BERMAN, P., FIAT, A., AND YAN, P. Online navigation in a room. In *SODA : selected papers from the third annual ACM-SIAM symposium on Discrete algorithms* (1994), Academic Press, Inc., pp. 319–341.

[3] BARSI, F., GRANDONI, F., AND MAESTRINI, P. A theory of diagnosability of digital systems. *IEEE Trans. Computers 25*, 6 (1976), 585–593.

[4] BENDER, M. A., FERNANDEZ, A., RON, D., SAHAI, A., AND VADHAN, S. The power of a pebble : exploring and mapping directed graphs. *Inf. Comput. 176*, 1 (2002), 1–21.

[5] BENDER, M. A., AND SLONIM, D. K. The power of team exploration : two robots can learn unlabeled directed graphs. In *Proceedings of the 35rd Annual Symposium on Foundations of Computer Science* (1994), IEEE Computer Society Press, Los Alamitos, CA, pp. 75–85.

[6] BETKE, M., RIVEST, R. L., AND SINGH, M. Piecemeal learning of an unknown environment. In *COLT '93 : Proceedings of the sixth annual conference on Computational learning theory* (New York, NY, USA, 1993), ACM Press, pp. 277–286.

[7] CZYZOWICZ, J., KOWALSKI, D., MARKOU, E., AND PELC, A. Searching for a black hole in tree networks. *Proc. 8th International Conference on Principles of Distributed Systems (OPODIS'2004)* (December 2004).

[8] CZYZOWICZ, J., KOWALSKI, D., MARKOU, E., AND PELC, A. Complexity of searching for a black hole. In *Fundamenta Informaticae 72* (2006), pp. 1–14.

[9] DENG, X., AND PAPADIMITRIOU, C. H. Exploring an unknown graph. *Journal of Graph Theory* (1999), 32 :265–297.

[10] DESSMARK, A., AND PELC, A. Optimal graph exploration without good maps. *Theor. Comput. Sci. 326*, 1-3 (2004), 343–362.

[11] DOBREV, S., FLOCCHINI, P., KRÁLOVIC, R., PRENCIPE, G., RUZICKA, P., AND SANTORO, N. Black hole search by mobile agents in hypercubes and related networks. In *OPODIS* (2002), A. Bui and H. Fouchal, Eds., vol. 3 of *Studia Informatica Universalis*, pp. 169–180.

[12] DOBREV, S., FLOCCHINI, P., PRENCIPE, G., AND SANTORO, N. Mobile search for a black hole in an anonymous ring. In *DISC '01 : Proceedings of the 15th International Conference on Distributed Computing* (London, UK, 2001), Springer-Verlag, pp. 166–179.

[13] Dobrev, S., Flocchini, P., Prencipe, G., and Santoro, N. Searching for a black hole in arbitrary networks : optimal mobile agent protocols. In *PODC '02 : Proceedings of the twenty-first annual symposium on Principles of distributed computing* (New York, NY, USA, 2002), ACM Press, pp. 153–162.

[14] Dobrev, S., Flocchini, P., Prencipe, G., and Santoro, N. Multiple agents rendezvous in a ring in spite of a black hole. In *OPODIS* (2003), M. Papatriantafilou and P. Hunel, Eds., vol. 3144 of *Lecture Notes in Computer Science*, Springer, pp. 34–46.

[15] Dobrev, S., Flocchini, P., and Santoro, N. Improved bounds for optimal black hole search with a network map. In *SIROCCO* (2004), R. Kralovic and O. Sýkora, Eds., vol. 3104 of *Lecture Notes in Computer Science*, Springer, pp. 111–122.

[16] Dudek, G., Jenkin, M., Milios, E., and Wilkes, D. Robotic exploration as graph construction. In *IEEE Transactions on Robotics and Automation* (1991), pp. 859–865.

[17] Duncan, C. A., Kobourov, S. G., and Kumar, V. S. A. Optimal constrained graph exploration. In *SODA '01 : Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms* (2001), Society for Industrial and Applied Mathematics, pp. 807–814.

[18] F.P. Preparata, G. M., and Chien, R. On the connection assignment problem of diagnosable systems. In *IEEE Transactions on Electronic Computers* (1967), pp. 848–854.

[19] Hromkovic, J., Klasing, R., Pelc, A., Ruzicka, P., and Unger, W. *Dissemination of information in communication networks.* Springer, 2005.

[20] Jansen, W., and Karygiannis, T. National institute of standards and technology special publication 800-19 - mobile agent security. Tech. rep., National Institute of Standards and Technology, August 1999.

[21] Klasing, R., Markou, E., Radzik, T., and Sarracco, F. Hardness and approximation results for black hole search in arbitrary graphs. In *SIROCCO* (2005), A. Pelc and M. Raynal, Eds., vol. 3499 of *Lecture Notes in Computer Science*, Springer, pp. 200–215.

[22] Lynch, N. A. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[23] Magic, G. Mobile agents white paper. In *General Magic Technology White Paper* (1998).

[24] Malek, M. A comparison connection assignment for diagnosis of multiprocessor systems. In *ISCA '80 : Proceedings of the 7th annual symposium on Computer Architecture* (New York, NY, USA, 1980), ACM Press, pp. 31–36.

[25] Markou, E., and Pelc, A. Efficient exploration of faulty trees. In *Proc. 15th Australasian Workshop on Combinatorial Algorithms (AWOCA'2004)* (Ballina Beach Resort, New South Wales, Australia, July 2004), pp. 52–63.

[26] Panaite, P., and Pelc, A. Exploring unknown undirected graphs. In *SODA '98 : Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms* (1998), Society for Industrial and Applied Mathematics, pp. 316–322.

[27] Panaite, P., and Pelc, A. Impact of topographic information on graph exploration efficiency. *Networks 36*, 2 (2000), 96–103.

[28] Pelc, A. Fault-tolerant broadcasting and gossiping in communication networks. *Networks 28*, 3 (1996), 143–156.

[29] Rao, N., Kareti, S., Shi, W., and Iyenagar, S. Robot navigation in unknown terrains : Introductory survey of non-heuristic algorithms, 1993.